

Siebel

Business Processes and Rules: Siebel Enterprise Application Integration

September 2023

Siebel

Business Processes and Rules: Siebel Enterprise Application Integration

September 2023

Part Number: F87442-01

Copyright © 1994, 2023, Oracle and/or its affiliates.

Authors: Siebel Information Development Team

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display in any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

The business names used in this documentation are fictitious, and are not intended to identify any real companies currently or previously in existence.

Contents

Preface	i
<hr/>	
1 What's New in This Release	1
What's New in Business Processes and Rules: Siebel Enterprise Application Integration, Siebel CRM 23.9 Update	1
What's New in Business Processes and Rules: Siebel Enterprise Application Integration, Siebel CRM 20.1 Update	1
What's New in Business Processes and Rules: Siebel Enterprise Application Integration, Siebel CRM 19.1 Update	1
2 Defining Workflows for Siebel EAI	3
Defining Workflows for Siebel EAI	3
Sample Integration Workflows	3
Testing the Workflow Integration Process	12
3 Creating and Using Dispatch Rules	15
Creating and Using Dispatch Rules	15
Overview of EAI Dispatch Service	15
Output Transformation	17
EAI Dispatch Service	19
Implementing EAI Dispatch Service	21
Testing Your EAI Dispatch Service Using Argument Tracing	24
Differences Between EAI Dispatch Service and Workflow	25
ProcessAggregateRequest Method	26
EAI Dispatch Service Scenarios	27
Examples of Search Expression Grammar	30
Examples of Dispatch Output Property Sets	32
4 Data Mapping Using the Siebel Data Mapper	35
Data Mapping Using the Siebel Data Mapper	35
Siebel Data Mapper Overview	35

EAI Data Mapping Engine	36
The Siebel Data Mapper	38
Creating Data Maps	39
Examples of Workflow Processes	42
EAI Data Mapping Engine Expressions	47
Addressing Fields in Components	49
Data Mapping Scenario	50
5 Data Mapping Using Scripts	51
Data Mapping Using Scripts	51
Overview	51
EAI Data Transformation	52
DTE Business Service Method Arguments	54
Map Functions	55
Data Transformation Functions	56
Siebel Message Objects and Methods	57
MIME Message Objects and Methods	75
Attachments and Content Identifiers in MIME Messages	79
XML Property Set Functions	80
EAI Value Maps	88
EAIGetValueMap Function	89
EAILookupSiebel Search Function	89
EAILookupExternal Search Function	89
CSSEIValueMap Translate Method	90
EAIGetValueMap unmappedKeyHandler Argument	90
EAIGetValueMap() Method	91
Exception Handling Considerations	92
Sample Siebel eScript	95
Conversion Between JSON and Siebel Property Sets	96

Preface

This preface introduces information sources that can help you use the application and this guide.

Using Oracle Applications

To find guides for Oracle Applications, go to the Oracle Help Center at <http://docs.oracle.com/>.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the [Oracle Accessibility Program website](#).

Contacting Oracle

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit [My Oracle Support](#) or visit [Accessible Oracle Support](#) if you are hearing impaired.

Comments and Suggestions

Please give us feedback about Oracle Applications Help and guides! You can send an email to: oracle_fusion_applications_help_ww_grp@oracle.com.

1 What's New in This Release

What's New in Business Processes and Rules: Siebel Enterprise Application Integration, Siebel CRM 23.9 Update

A new section has been added to this guide for this release - *Conversion Between JSON and Siebel Property Sets*.

What's New in Business Processes and Rules: Siebel Enterprise Application Integration, Siebel CRM 20.1 Update

No new features have been added to this guide for this release. This guide has been updated to reflect only product name changes.

What's New in Business Processes and Rules: Siebel Enterprise Application Integration, Siebel CRM 19.1 Update

No new features have been added to this guide for this release. This guide has been updated to reflect only product name changes.

2 Defining Workflows for Siebel EAI

Defining Workflows for Siebel EAI

This chapter explains workflow integration processes and how to use them to develop your integration projects. The chapter depends on several sample workflows that are included in Siebel. It contains the following topics:

- *Sample Integration Workflows*
- *Testing the Workflow Integration Process*

Note: In version 8.0 and higher of Siebel, workflow processes are created in Siebel Tools. However, the sample workflows in this chapter are found in the Administration - Business Process screen, then the Workflow Processes view in the Siebel client.

For information on creating workflow processes, see *Siebel Business Process Framework: Workflow Guide*.

Sample Integration Workflows

Siebel EAI includes several sample workflows that illustrate how you can receive, process, and send integration messages. This chapter includes four of those samples, along with brief descriptions that are intended to help you understand the workflow elements specific to Siebel EAI.

Note: One of the methods of invoking a workflow process is through a workflow policy. To invoke a workflow process that includes steps that call EAI adapters from a workflow policy, you must create a workflow policy action using the Run Workflow Process workflow policy program. The workflow policy action will invoke the Workflow Process Manager component. For information on creating workflow policies, see *Siebel Business Process Framework: Workflow Guide*.

The sample workflows explained in this chapter include:

- *Import Account (File)*
- *Export Account (File)*
- *Import Employee (MQSeries)*
- *Export Employee (MQSeries)*

Import Account (File)

This is a sample workflow process that reads an XML file (c:\account.xml) and imports the account information into the Siebel environment using the EAI XML Read from File business service. The EAI XML Read from File business service converts the data and the EAI Siebel Adapter updates the Siebel database.

To review the Import Account (File) sample workflow process

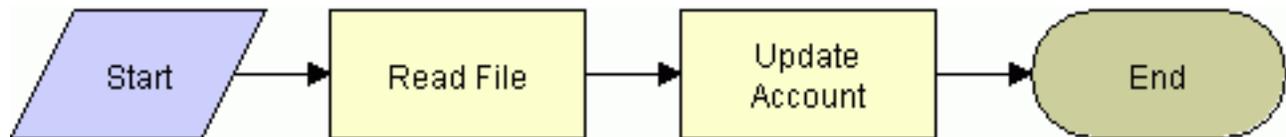
1. Navigate to Administration - Business Process, then Workflow Processes.

2. Query for Import Account (File).
3. With the Import Account (File) workflow process selected, click the Process Properties tab in the lower applet to review the process properties for this workflow process.

Workflow process properties are global to the entire workflow. For example, as shown in the following table, the Import Account (File) workflow has several properties. The Account Message is defined to identify the output of the Read File step (a parsed version of the XML Account Message) as a hierarchical structure. The Error Message, Error Code, Object Id, and Siebel Operation Object Id properties are included in the workflow by default.

Name	Data Type	In/Out
Account Message	Hierarchy	In/Out
Error Code	String	In/Out
Error Message	String	In/Out
Object Id	String	In/Out

4. Click the Process Designer tab in the lower applet to review the process design for this workflow process.



5. Double-click the Read File step to review its method and arguments.

This step uses the Read Siebel Message method of the EAI XML Read from File business service to convert XML from a file into an integration object hierarchy, with the following input argument.

Input Arguments	Type	Value
File Names	Literal	c:\account.xml

Note how the path and file name are specified as a string in the Value field of the Input Arguments applet. Also note the following output property for this step.

Property Name	Type	Output Argument
Account Message	Output Argument	Siebel Message

6. Double-click the Update Account step to review its method and arguments.

This step uses the EAI Siebel Adapter business service with the Insert or Update method to read the Siebel Message and update or insert the Account object in the Siebel Database. The EAI Siebel Adapter uses the information in the XML file and the following input argument to accomplish this task.

Arguments	Type	Property Name	Property Data Type
Siebel Message	Process Property	Account Message	Hierarchy

Because the Insert or Update method is specified on the EAI Siebel Adapter business service, this step checks the Siebel Database to see if the Account object defined in the XML file already exists in the database. If the account exists, then it updates the account in the database with the account instance from the XML file; otherwise, it inserts the account into the database.

Export Account (File)

This sample workflow process exports an account to a file in an XML format. This workflow uses the EAI Siebel Adapter and the EAI XML Write to File business service to query the data and then convert the data from the Siebel business object to an XML document.

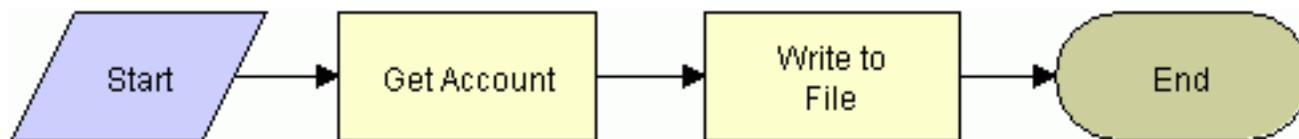
To review the Export Account (File) sample workflow process

1. Navigate to Administration - Business Process, then Workflow Processes.
2. Query for Export Account (File).
3. With the Export Account (File) workflow process selected, click the Process Properties tab to review the process properties for this workflow process.

Workflow process properties are global to the entire workflow. For example, as shown in the following table, the Export Account (File) workflow has several properties. The Output Message property is defined to identify the outbound account as a hierarchical structure. The Object Id property is included in each workflow by default.

Name	Data Type	In/Out
Object Id	String	In/Out
Output Message	Hierarchy	In/Out

4. Click the Process Designer tab in the lower applet to review the process design for this workflow process.



5. Double-click the Get Account step to review its method and arguments.

This step uses the EAI Siebel Adapter business service to query an account from your Siebel Database. The EAI Siebel Adapter uses the following input arguments.

Input Arguments	Type	Value	Property Name	Property Data Type
Output Integration Object Name	Literal	Sample Account	-	-
Object Id	Process Property	-	Object Id	String

Note that Output Integration Object Name of Sample Account is part of the query criteria. The Sample Account integration object describes the structure of the Account business object and was created using the Integration Object Builder. The other part of the query criteria is the Object Id, which is a process property that includes the account number 1-6 defined as a process property before.

Also note the following output property for this step.

Property Name	Type	Output Argument
Output Message	Output Argument	Siebel Message

The output from this step is Output Message. Output Message is a process property that will include the Siebel Message, which contains data for the account. The format is specified by the Sample Account integration object.

6. Double-click the Write to File step to review its method and arguments.

This step invokes the EAI XML Write to File business service with the Write Siebel Message method. The EAI XML Write to File uses the following input arguments.

Input Arguments	Type	Value	Property Name	Property Data Type
File Name	Literal	c:\account.xml	-	-
Siebel Message	Process Property	-	Output Message	Hierarchy

The EAI XML Write to File business service converts the hierarchical message to XML and writes the resulting document to the file named in the File Name argument.

Import Employee (MQSeries)

This is a sample workflow process that receives an XML string from an IBM MQSeries queue and updates the Employee instance in the Siebel Database.

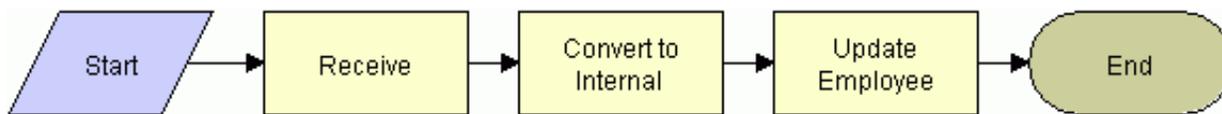
To review the Import Employee (MQSeries) sample workflow process

1. Navigate to Administration - Business Process, then Workflow Processes.
2. Query for Import Employee (MQSeries).
3. With the Import Employee (MQSeries) process selected, click the Process Properties tab to review its process properties.

Workflow process properties are global to the entire workflow. For example, as shown in the following table, the Import Employee (MQSeries) workflow has several properties. The Employee Message contains the object as an integration object hierarchy, when converted. The object must be in that format before it can be inserted or updated in the Siebel environment. The Employee XML property defines the MQSeries message as XML recognizable by Siebel applications. The Error Code, Error Message, Object Id, and Siebel Operation Object Id properties are included in the workflow by default.

Name	Data Type	
Employee Message	Hierarchy	In/Out
Employee XML	Binary	In/Out
Error Code	String	In/Out
Error Message	String	In/Out
Object Id	String	In/Out
Siebel Operation Object Id	String	In/Out

- Click the Process Designer tab in the lower applet to review the process design for this workflow process.



Note: When using the MQSeries Receiver, remember that the MQ Receiver task will read the message from the queue and pass it into your workflow process in the <Value> field. This means your workflow process does not need to read the message from the MQSeries Queue. To get the XML string that has been read, you need to create a process property and set its default value as follows: Name=MyXMLStringProperty and Default=<Value>. You should use this process property as the input to the EAI XML Converter business service.

- Double-click the Receive step to review its method and arguments.

This step uses the Receive method of the EAI MQSeries Server Transport to get the inbound message from the Employee physical queue named in the Physical Queue Name argument. This step uses the following input arguments.

Input Arguments	Type	Value
Physical Queue Name	Literal	Employee
Queue Manager Name	Literal	Siebel

As shown in the following table, the output from this step is put into the Employee XML process property with the assumption that the inbound message is already in XML format.

Property Name	Type	Output Argument
Employee XML	Output Argument	Message Text

- Double-click the Convert to Internal step to review its method and arguments.

This step uses the XML to Property Set method of the EAI XML Converter to convert the inbound message to the Siebel business object format. The step uses the following input argument.

Input Arguments	Type	Property Name	Property Data Type
XML Document	Process Property	Employee XML	Binary

Input Arguments	Type	Property Name	Property Data Type

The output from this step is passed in the Employee Message output argument as described in the following table.

Property Name	Type	Output Argument
Employee Message	Output Argument	Siebel Message

7. Double-click the Update Employee step to review its method and arguments.

This step uses the EAI Siebel Adapter business service with the Insert or Update method and the following input argument to update the database.

Input Arguments	Type	Property Name	Property Data Type
Siebel Message	Process Property	Employee Message	Hierarchy

The EAI Siebel Adapter checks the Siebel Database for an Employee record that matches the current instance of Employee in the Employee Message property. If an Employee record matching the current instance does not exist in the database, the EAI Siebel Adapter inserts the record into the database; otherwise, it updates the existing record with the instance.

Export Employee (MQSeries)

This is a sample workflow process that sends an XML string for an employee to an IBM MQSeries queue.

To review the Export Employee (MQSeries) sample workflow process

1. Navigate to Administration - Business Process, then Workflow Processes.
2. Query for Export Employee (MQSeries).
3. With the Export Employee (MQSeries) workflow process selected, click the Process Properties tab to review the process properties defined for this workflow process.

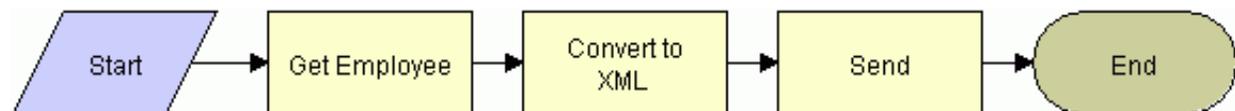
Workflow process properties are global to the entire workflow. For example, as shown in the following table, the Export Employee (MQSeries) workflow has multiple properties. The Employee Message contains the object as an integration object hierarchy, before conversion. The Employee XML property specifies the Siebel object

that has been converted to XML. The Error Code, Error Message, Object Id, and Siebel Operation Object Id properties are included in each workflow by default.

Name	Data Type	In/Out	Default String
Employee Message	Hierarchy	In/Out	-
Employee XML	Binary	In/Out	-
Error Code	String	In/Out	-
Error Message	String	In/Out	-
Object Id	String	In/Out	1-548
Siebel Operation Object Id	String	In/Out	-

Note that the Object Id process property is set to 1-548 in the Default String column. This string identifies an actual employee record in the Siebel Database by its Row Id. You can set this workflow to use the active employee record instead of specifying a hard-coded employee number. You can accomplish this by creating a button that invokes this workflow from the Administration - User, then the Employees view, or you can pass the value of the Object Id into the workflow process as an input argument.

- Click the Process Designer tab in the lower applet to review the process design for this workflow process.



- Double-click the Get Employee step to review its method and arguments.

This step uses the Query method of the EAI Siebel Adapter business service, with the following input arguments to get an instance of an Employee record from the Siebel Database. The Sample Employee integration object describes the structure of the Employee business object and was created using the Integration Object Builder wizard. The other part of the query criteria is the Object Id, which is a process property containing value 1-548.

Input Arguments	Type	Value	Property Name	Property Data Type
Output Integration Object Name	Literal	Sample Employee	-	-
Object Id	Process Property	-	Object Id	String

The output from this step is passed in the Employee Message output argument as described in the following table.

Property Name	Type	Output Argument
Employee Message	Output Argument	Siebel Message

6. Double-click the Convert to XML step to review its method and arguments.

This step uses the Property Set to XML method of the EAI XML Converter business service to convert the outbound Siebel Message to XML. The converter stores the outbound Siebel Message in the Employee XML output argument with the following input argument.

Input Arguments	Type	Property Name	Property Data Type
Siebel Message	Process Property	Employee Message	Hierarchy

The output from this step is passed in to the Employee XML output argument as shown in the following table.

Property Name	Type	Value	Output Argument
Employee XML	Output Argument	-	XML Document

7. Double-click the Send step to review its method and arguments.

This step invokes the EAI MQSeries Server Transport business service with the Send method to put the XML message onto the MQSeries queue, Employee. The message is represented by the Message Text argument, as shown in the following table.

Input Arguments	Type	Value	Property Name	Property Data Type
Message Text	Process Property	-	Employee XML	String
Physical Queue Name	Literal	Employee	-	-
Queue Manager Name	Literal	Siebel	-	-

The Queue Manager that handles the request is called Siebel. The XML message is put onto the Employee queue, where it remains until another application retrieves it from the queue.

Testing the Workflow Integration Process

When you have finished defining your integration workflow process, you can use the Workflow Process Simulator to test its behavior.

The Workflow Process Simulator, included in the Workflow Process Manager, allows you to validate your processes before deploying them in production environments.

When you simulate an integration process that performs some external action—for example, the Export Account (File) workflow writes an XML file to a disk location—you can verify the end result by checking if the output object exists, or if a predetermined event has occurred.

Note: You can also enable detailed client logging and use the `/s` option for creating SQL spool scripts. This option provides more detailed information when running the integration workflow process in the Workflow Process Simulator. For details, see *Siebel Remote and Replication Manager Administration Guide*.

To test the workflow processes described in this chapter using the Workflow Process Simulator, you must export them to an XML file from the Workflow Processes view in the Siebel client, and then import the XML file into the Workflow Processes list in Siebel Tools. The Workflow Process Simulator runs from Siebel Tools.

Exporting the Workflow Process to an XML File

You export the workflow process from the Workflow Processes view in the Siebel client. In this example, Export Account (File) is exported.

To export a workflow process to an XML file

1. Navigate to Administration - Business Process, then Workflow Processes.
2. Query for Export Account (File).
3. From the pull-down menu, choose Export Workflow.
4. Choose a location to save the XML file.

The workflow process XML file, in this example Export Account (File).xml, is created.

Importing the XML File Into Siebel Tools

You import the workflow process XML file into the Workflow Processes list in Siebel Tools.

To import the XML file into Siebel Tools

1. In the Object Explorer in Siebel Tools, select Workflow Process.
2. Right-click inside the Workflow Processes list in the Object List Editor, then choose Import Workflow Process.
3. Browse for the workflow process XML file, then click Open.

The workflow process is imported into Siebel Tools.

Running the Workflow Process Simulator

You run the Workflow Process Simulator from the Workflow Process list in Siebel Tools.

To simulate a workflow process

1. In Siebel Tools, choose Toolbars, then the Simulate view to activate the Simulator.
2. In the Object Explorer in Siebel Tools, select Workflow Process.
3. Query for Export Account* in the Object List Editor.
4. Right-click on Export Account (File), then choose Simulate Workflow Process.

The Workflow Process Designer appears and the Start element has a highlighted border to indicate that it is the active element.

5. Click Start Simulation in the Simulate toolbar.

A new instance of the Siebel Debugger starts and then runs the Simulator.

6. Click Simulate Next to activate the next step.

As you step through the process, the border of each active element becomes highlighted in turn, unless the simulator encounters an error, in which case it displays an error message alert.

7. Click Complete Simulation to pause.
8. Click Stop Simulation to stop.

For more information about running the Workflow Process Simulator, reviewing process values, and using Workflow Process Manager and Workflow Batch Manager, see *Siebel Business Process Framework: Workflow Guide*.

Note: Use the Workflow Process Simulator only for testing purposes. Do not use the Workflow Process Simulator to load data.

3 Creating and Using Dispatch Rules

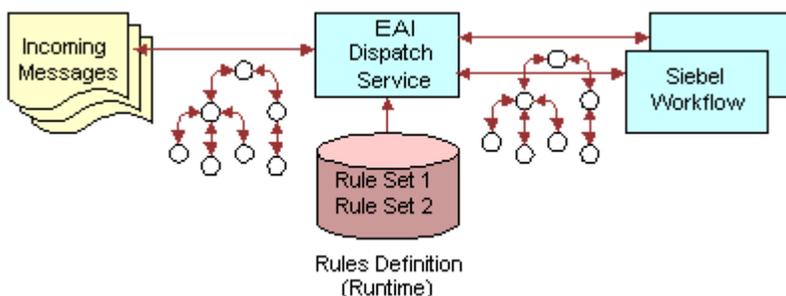
Creating and Using Dispatch Rules

This chapter gives an overview on the EAI Dispatch Service, transforming output, and implementing a new dispatch service. It contains the following topics:

- *Overview of EAI Dispatch Service*
- *Output Transformation*
- *EAI Dispatch Service*
- *Implementing EAI Dispatch Service*
- *Testing Your EAI Dispatch Service Using Argument Tracing*
- *Differences Between EAI Dispatch Service and Workflow*
- *ProcessAggregateRequest Method*
- *EAI Dispatch Service Scenarios*
- *Examples of Search Expression Grammar*
- *Examples of Dispatch Output Property Sets*

Overview of EAI Dispatch Service

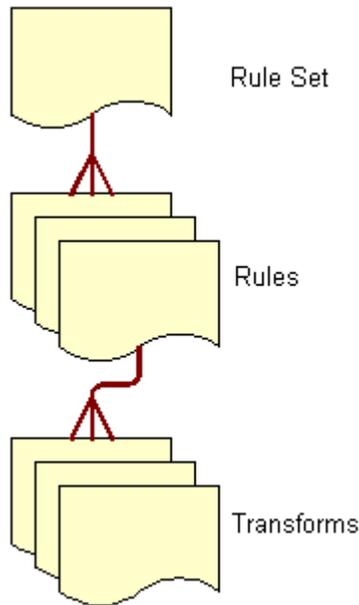
The EAI Dispatch Service is a rule-based dispatching business service that invokes business services based on the properties of its input property set. The EAI Dispatch Service can execute transformations on an input property set before dispatching it to the target business service. Such transformations can be useful for setting business service arguments or workflow process properties. They can also be used to do limited hierarchy manipulation such as discarding the envelope of an XML document. The following illustrates the EAI Dispatch Service process.



Although the EAI Dispatch Service is a utility to invoke one business service from another business service based on specified rules, one of its primary uses is to accomplish inbound and outbound integration. The EAI Dispatch Service can be the first business service of the inbound integration to decide which business service should process an incoming document. It can also be the last step of the outbound integration to send the outgoing document to the right transport. The EAI Dispatch Service is similar to the branching in Siebel Workflow. To determine whether to use Siebel Workflow or the EAI Dispatch Service, see *Differences Between EAI Dispatch Service and Workflow*.

EAI Dispatch Service Rule Hierarchy

The EAI Dispatch Service has a three-layer rules hierarchy as illustrated in the following figure.



Rule Sets

Rule Sets are sets of rules that you define in a particular sequence. EAI Dispatch Service parses the input document using these rules in sequence until it finds a rule that matches the input.

Rules

Rules are individual entities in a rule set. Each rule consists of data transformations, search expression grammar, and zero or more rule transforms. You define rules by using search expression grammar to establish how you want an input message to be routed. For details, see [Search Expression Grammar](#).

Data Transformation

A transform specifies how the intermediate output is going to be generated before it is dispatched to the service and the method you specified in the rule. For details, see [Output Transformation](#).

EAI Dispatch Service Methods

EAI Dispatch Service uses the methods described in the following table.

Method	Description
Dispatch	This method parses the input against the rules and dispatches it to the appropriate business service and business service method for further processing.

Method	Description
Lookup	This method returns the intermediate output generation as specified by the rule output properties without dispatching it to any business service. You use this method for debugging purposes, as well as manipulating property sets within business service or workflow.
ProcessAggregateRequest	This method allows multiple invocations of business services in a single request. The output for each request will be combined into a single Siebel property set or XML document. The input to this method is an XML document. For details, see ProcessAggregateRequest Method .
Purge	The Purge method clears any data that has been cached by the EAI Dispatch Service and does not take in any input arguments.

The EAI Dispatch Service executes the following at run time:

- Matches the input with a dispatch rule.
- Evaluates the transforms.
- Dispatches the output to a business service if the method is set to Dispatch.

Search Expression Grammar

Search expression grammar is used by the EAI Dispatch Service to parse incoming messages and determine the course of action. Search expression grammar is based on the XPath standard. The following information presents the definitions you use to construct a search expression.

Symbols	Description
/	A forward slash indicates a new level in the hierarchy. The first slash indicates the root of the hierarchy.
@	An at symbol indicates the attribute.
*	An asterisk indicates no specific criteria and that everything matches in the input. Asterisks cannot be used with attributes.
Name	This is the literal value for which the EAI Dispatch Service searches the document.

Note: See [Examples of Search Expression Grammar](#) for additional information and examples.

Output Transformation

Before dispatching the incoming hierarchy to the business service, EAI Dispatch Service can be used to perform some transformations to the hierarchy to make it appropriate for the target business service. A transform specifies how the

intermediate output, in the memory, is going to be generated before it is dispatched to the service and the method you specified in the rule.

If you do not define any transforms, the EAI Dispatch Service will send the input directly to the business service. However, if you define transforms, the EAI Dispatch Service will create intermediate output based on the values of the transforms before sending the input to the business service you have defined in your rule.

Transforms are specified using one or more of the following targets in permissible combination.

RootHierarchy

This target creates a new output root hierarchy based on the source expression. The source expression specifies a node in the input hierarchy. The hierarchy rooted at this node is copied as the target root hierarchy. You can use the root hierarchy for minor modifications, such as adding a property, to the input hierarchy.

Only one root hierarchy transform can be specified because this transform always creates a new hierarchy. The root hierarchy transform is always executed before any other transforms in the combination.

Note: For the following targets, if an output hierarchy does not exist at the time of invoking the target, an output hierarchy is first created with just an empty root node before the target is applied.

ChildHierarchy

This target creates a new hierarchy as a child of the current output root hierarchy, based on the source expression. The source expression specifies a node in the input hierarchy. The hierarchy rooted at this node is copied as a new child hierarchy. You can use the child hierarchy for adding service arguments to an incoming document before dispatching to workflow or business service.

Type

This target sets the Type field to Source Expression in the root node of output hierarchy.

Value

This target sets the Value field to Source Expression in the root node of output hierarchy.

Property

This target creates or overwrites a property with name Property Name and value Source Expression in the root node of output hierarchy. You can use property to add business service arguments or workflow process properties.

As described in the following table, for certain targets, in addition to the dispatch grammar, literal values can be used for the Source Expression property to retrieve the data from the input message.

Target	Source Expression	Property Name
Property	Dispatch grammar or a literal value enclosed in quotes to search for a value	Name of the Property
ChildHierarchy	Grammar to search for the hierarchy	N/A
RootHierarchy	Grammar to search for the hierarchy	N/A

Target	Source Expression	Property Name
Type	Dispatch grammar or Literal value enclosed in quotes to search for a value	N/A
Value	Literal value enclosed in quotes	N/A

Note: You can combine one or more of the preceding transforms to achieve the desired transformation. The combination should not include more than one Root Hierarchy transform, Type transform, or Value transforms. However, it can include multiple Property transforms, as long as the names of the properties are different. Also if you do not want to transform the input data, but need to add an entry in transform—for example the process name of the dispatching workflow, you have to add another entry to the transform with Target: RootHierarchy, Source Expression: /*, and no Property Name. If you do not have a RootHierarchy transform, an empty PropertySet will be created and the called dispatching service will receive an invalid hierarchy data.

EAI Dispatch Service

You can use the EAI Dispatch Service to:

- Respond to a request from an external system. This can be a request to query data or a request to insert data into the Siebel Database. See *Inbound Requests*.
- Send data to an external system based on an event in Siebel applications. See *Outbound Requests*.

The EAI Dispatch Service works with the hierarchy in the property set, which may be in some cases different from the hierarchy in your document. When dispatching XML documents, you should use the XML Hierarchy Converter because it generates a hierarchy matching the hierarchy in the XML document.

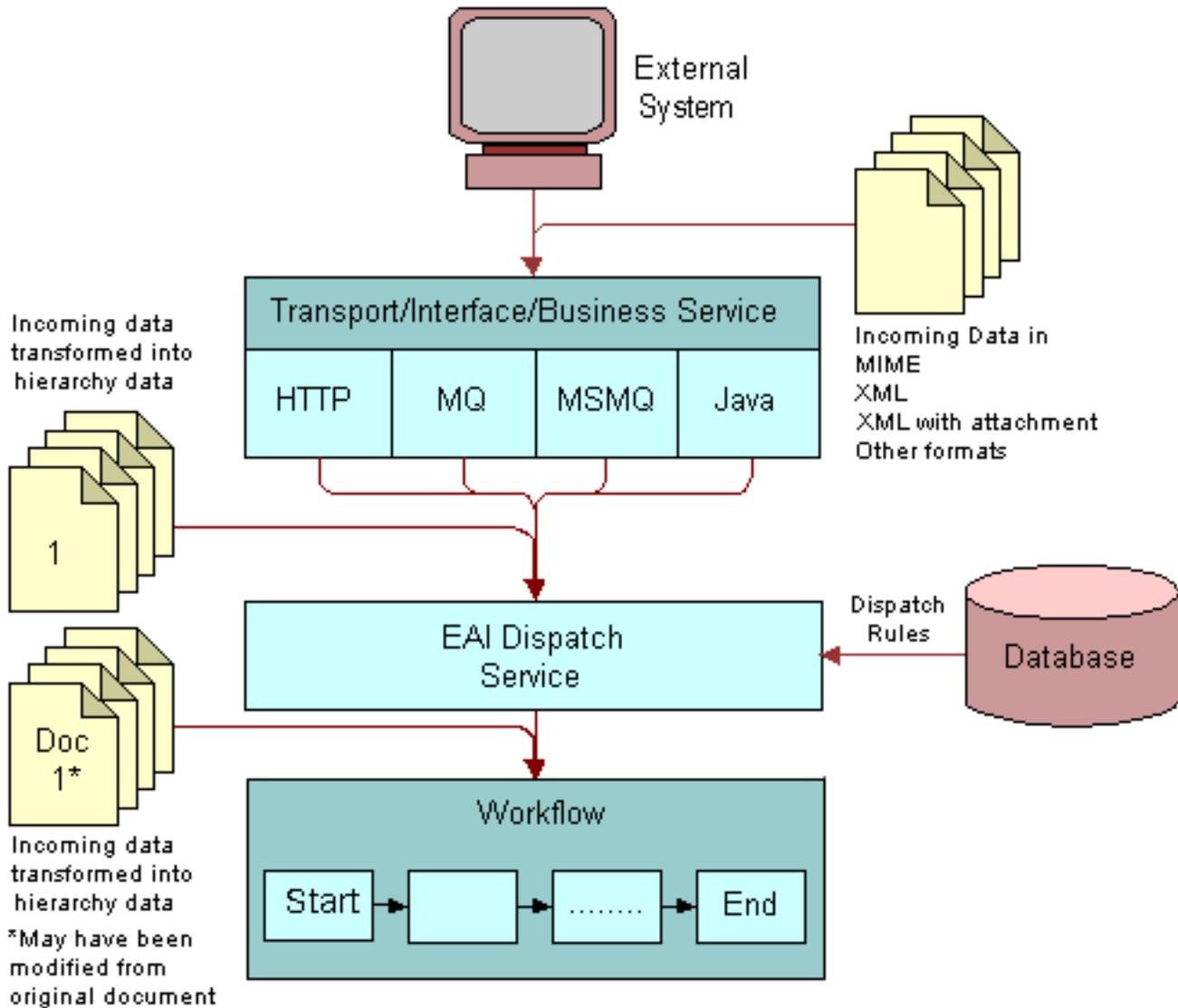
Note: For details on the XML Hierarchy Converter, see *XML Reference: Siebel Enterprise Application Integration*.

Use the business service argument tracing facility provided by the EAI Dispatch Service to understand the input property set hierarchy. This facility dumps the input and the output of the EAI Dispatch Service as XML. For details, see *Testing Your EAI Dispatch Service Using Argument Tracing*.

Inbound Requests

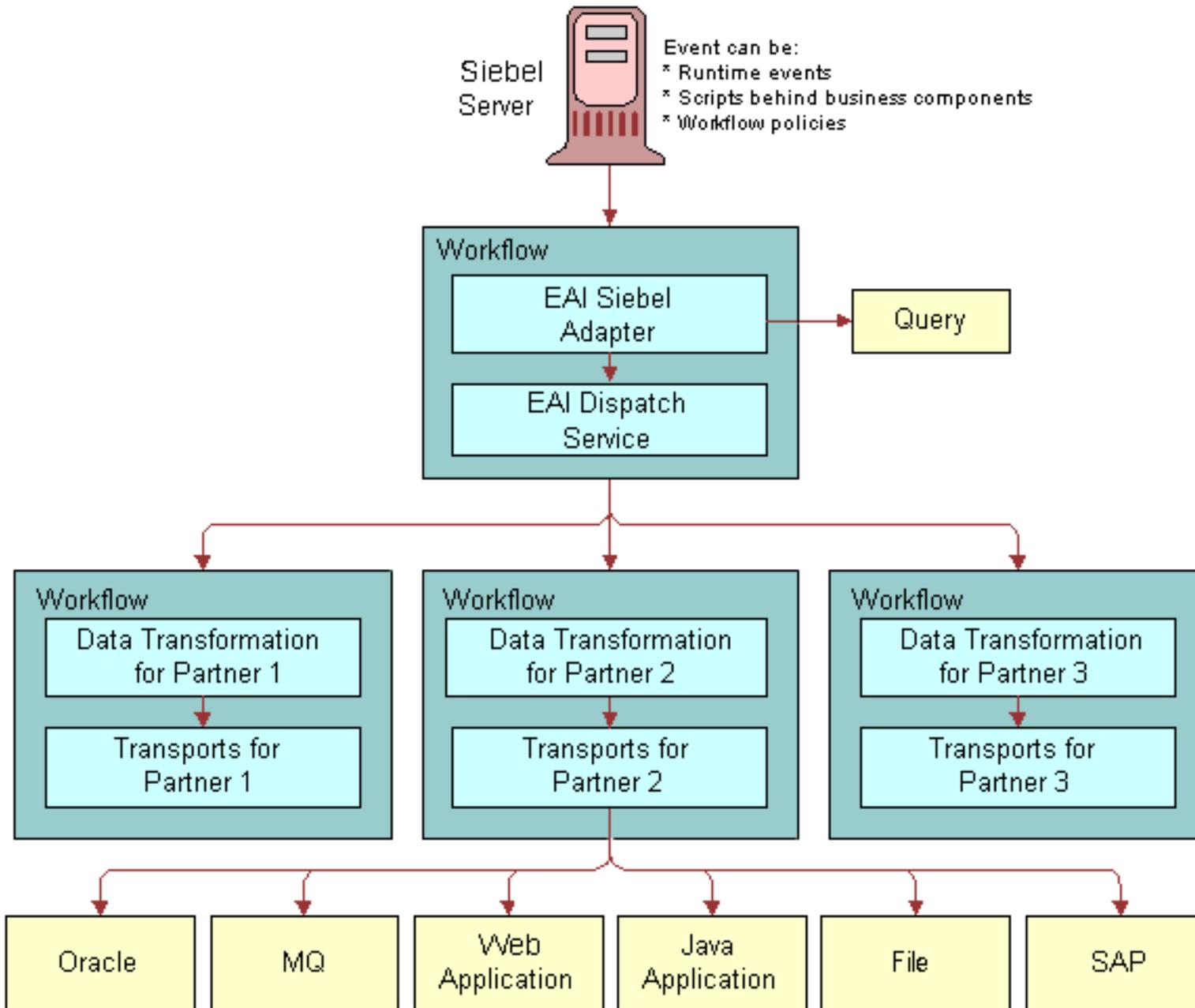
The steps for creating an inbound or an outbound EAI Dispatch Service are very similar, as illustrated in the following figures.

The following illustrates the high-level architecture of an inbound EAI Dispatch Service.



Outbound Requests

The steps for creating an outbound EAI Dispatch Service are the same as the steps for an inbound EAI Dispatch Service with some differences in the workflow. The following illustrates the high-level architecture of an outbound Dispatch Service. For details on how to create an outbound workflow, see [Outbound Scenario](#).



Implementing EAI Dispatch Service

The following checklist lists the steps you need to take to implement a new EAI Dispatch Service. These steps are the same whether an external system is requesting data from a Siebel application, or inserting data into a Siebel application, or when a Siebel application sends a request to an external system.

	Checklist
#	Create a workflow to be called by the EAI Dispatch Service. For details, see <i>Creating a Workflow</i> .
#	Define a Rule Set. For details, see <i>Defining Rule Sets</i> .
#	Define Rules. For details, see <i>Defining Rules</i> .
#	Define Transforms. For details, see <i>Defining Transforms</i> .
#	Set up the EAI Dispatch Service to invoke the workflow. For details, see <i>Creating a Workflow</i> .
#	Test your EAI Dispatch Service. For details, see <i>Testing Your EAI Dispatch Service Using Argument Tracing</i> .

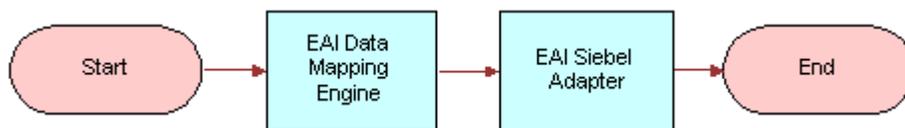
Creating a Workflow

Design a workflow process to be called by EAI Dispatch Service upon receiving a request from an external system.

Note: For details on how to use Workflow Process Manager, see *Siebel Business Process Framework: Workflow Guide*.

To design a workflow to receive a request from an external system

1. In Siebel Tools, set up a workflow process to include the following steps: Start, EAI Data Mapping Engine, EAI Siebel Adapter, End.



2. Create process properties to pass incoming data from the EAI Dispatch Service.

Because you have to pass data (as a hierarchy) from the EAI Dispatch Service to the workflow, you need to create a process property of type Hierarchy to receive this data. The name of the property should match the

root tag of the hierarchy you are passing. If you use XML Hierarchy Converter with the EAI Dispatch Service, then you use the property XMLHierarchy.

Also, you may want to pass other parameters, such as what data map to use, from the EAI Dispatch Service. Create process properties of type String to receive such parameters. The name of the property should match the Property Name used in your dispatch transform.

Defining Rule Sets

Rule sets are used by the EAI Dispatch Service to search the incoming data for specific criteria.

To define a rule set

1. Navigate to Administration - Integration, then the EAI Dispatch Service View.
2. Click New on the Rule Sets list applet to create a new rule set.
3. Give this rule set a meaningful name such as AribaAccountToSiebel.
4. Save the rule set.

Defining Rules

This topic describes how to define rules.

To define rules

1. Click New on the Rules list applet on the EAI Dispatch Service View.
2. Provide the following fields for this record:
 - o **Sequence.** Enter a sequence number. This determines the sequence in which the application evaluates the rules.
 - o **Search Expression.** Actual logic behind what the rule is looking for in the input. Define the Search Expression using Dispatch Rule Grammar. For details, see [Search Expression Grammar](#).
 - o **Property Value (Optional).** Populate this field with the value for the property that the input is to be matched with.
 - o **Dispatch Service.** The business service that you want to dispatch the input to. You leave this blank if you intend to use the Lookup method.
 - o **Dispatch Method.** Pick a method for the business service you defined in the Dispatch Service field.
3. Save your rules.

The system validates search expression grammar. If you have not set your rules properly, you will receive an error message. See [Examples of Search Expression Grammar](#) for examples of valid search expressions.

Defining Transforms

This topic describes how to define transforms.

To define transforms

1. Click New on the Transforms list applet on the EAI Dispatch Service View to create a new transform.
2. Provide the following fields for the new record:
 - o **Target.** Defines how the intermediate output is going to be generated before it is dispatched to the service and the method you specified in the rule. For details, see [Output Transformation](#).
 - o **Source Expression.** The source expression is used to assign a value to the target. You can either use a search expression pointing to a node in the input hierarchy or a literal value enclosed in quotes. For details, see [Search Expression Grammar](#).
 - o **Property Name.** The name of the property to be set. This value is only used when the Target is set to Property. For the other Target types this field is inactive.

Note: See [EAI Dispatch Service Scenarios](#) and [Examples of Search Expression Grammar](#) for more details on these parameters.

3. Save your transform.

This saves and validates your transform.

Invoking a Workflow Process From an EAI Dispatch Service

Once you created your workflow, you need to set up your EAI Dispatch Service to invoke it.

To invoke a workflow process with an EAI Dispatch Service

1. Navigate to Administration - Integration, then the EAI Dispatch Service View.
2. Select the target rule set.
3. Select the rule that invokes the workflow process.
4. For the selected rule set the following values:
 - o **Dispatch Service.** Workflow Process Manager
 - o **Dispatch Method.** Execute Process
5. For the selected rule insert a new record in the Transforms applet and fill in the following values:
 - o **Target.** Property. You can select the Property value from a list of values.
 - o **Source Expression.** Name of the workflow process to run. Make sure you include double quotes around the name, for example, "my workflow process".
 - o **Property Name.** Process Name. You can select the Property Name value from a list applet.

Testing Your EAI Dispatch Service Using Argument Tracing

You should use the Business Service Simulator to test your EAI Dispatch Service before using it in your production environment. You can use argument tracing to write the input and the output of the EAI Dispatch Service as XML.

Note: For details on how to use the Business Service Simulator, see *Integration Platform Technologies: Siebel Enterprise Application Integration*.

To use the EAI dispatch service argument tracing

1. Set the server parameter EnableServiceArgTracing to true.
2. Set the appropriate event level for EAIDispatchSvcArgTrc on your server component:
 - o **Event level 3.** Leads to input arguments being written out when errors occur.
 - o **Event level 4.** Leads to both input and output being written out.

If arguments are written out, there will be a trace log entry indicating the filename in the log directory. The filenames will have the following form:

```
service name_input or output
_arggs_
a big number
.dmp
```

For example:

```
EAIDispatchService_input_arggs_270613751.dmp
```

Note: To open the file in a XML editor, you can rename the extension to XML.

Differences Between EAI Dispatch Service and Workflow

Although the EAI Dispatch Service is very similar to Siebel Workflow in initiating a task based on a condition, there are some limitations in Siebel Workflow that you can overcome using the EAI Dispatch Service. Siebel Workflow operates on business components as opposed to property sets, so Siebel Workflow can only branch based on fields in a business component. Furthermore, with Siebel Workflow you cannot route incoming documents based on property sets, because the workflow decision points cannot search inside of arbitrary property sets.

The following table provides some guidance to help you determine the best method for your business requirements.

Requirements	EAI Dispatch Service	Workflow	Notes
Need to route the incoming document based on its structure or content	Yes		The EAI Dispatch Service can route incoming documents based on property sets, whereas workflow can only branch based on fields in a business component.
Multiple dispatch targets	Yes		The EAI Dispatch Service is a better choice because writing a workflow to include every branch can be

Requirements	EAI Dispatch Service	Workflow	Notes
			unwieldy, but you can have many EAI Dispatch Service rules.
Need to change input property set before dispatching	Yes		The EAI Dispatch Service is the better choice since it has more powerful mapping capabilities than workflow.
Need more complex processing on the input message before dispatching		Yes	The EAI Dispatch Service can branch based on the content of the input document, whereas workflow can branch based on business service.
Workflow options are sufficient for your requirements		Yes	In this case, Siebel Workflow is the best choice.

ProcessAggregateRequest Method

The ProcessAggregateRequest method allows you to perform multiple invocations of business services in a single request. The method bundles the output for each request into a single Siebel property set or XML document.

When using the ProcessAggregateRequest method with the EAI Dispatch Service business service, you need to define an input argument called AggregatedServiceRequest, with type Hierarchy for the EAI Dispatch Service to use to store the incoming data.

The following example is the input argument for this method, using XML to represent the PropertySet.

```
....
<PropertySet>
  <AggregatedServiceRequest>
```

This is the input/output method argument for the ProcessAggregatedRequest method. The EAI Dispatch Service with ProcessAggregateRequest Method looks for this XML tag within the XML document to determine where it needs to start reading the document.

```
<BusinessServiceWrapper
```

wrapper around the business service. The name of the wrapper has no effect on the EAI Dispatch Service.

```
  BusinessServiceName=...
```

XML tag for business service

```
    BusinessServiceMethod=...>
```

XML tag for business service method

```
<ArgumentWrapper
```

wrapper around the business service arguments. The name of the wrapper has no effect on the EAI Dispatch Service.

```
  XMLTagArgument1=...
```

XML tag for the first argument. Replace this tag with the correct XML tag for the argument your business service method is using.

```
XMLTagArgument2=...
```

XML tag for the second argument. Replace this tag with the correct XML tag for the argument your business service method is using.

```
... />  
</BusinessServiceWrapper>
```

Note: For examples, see *Outbound Scenarios Using ProcessAggregateRequest*.

EAI Dispatch Service Scenarios

The following business scenarios explain how you might accomplish commonly performed tasks using the EAI Dispatch Service.

Outbound Scenario

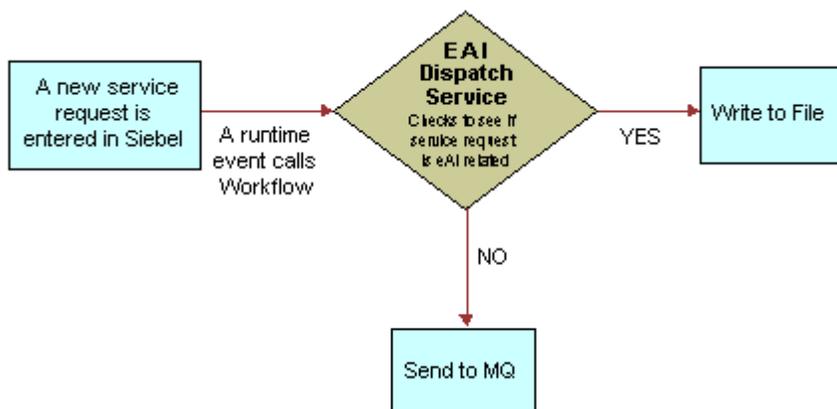
For this scenario, you want to dispatch a service request as soon as it is created. The scenario assumes that:

- You are only interested in service requests logged against EAI.
- You know how to design a workflow that gets triggered as a new service request is created.

Note: There are number of different ways to trigger a workflow process. For details, see *Siebel Business Process Framework: Workflow Guide*.

- You want the other non-EAI service requests to be sent to an MQSeries.

The following illustrates this scenario.



To create this scenario

1. Create a rule set with a search expression to check if the Service Request Area is set to EAI.
2. Create a workflow that is triggered when the criterion defined in Step 1 is matched.

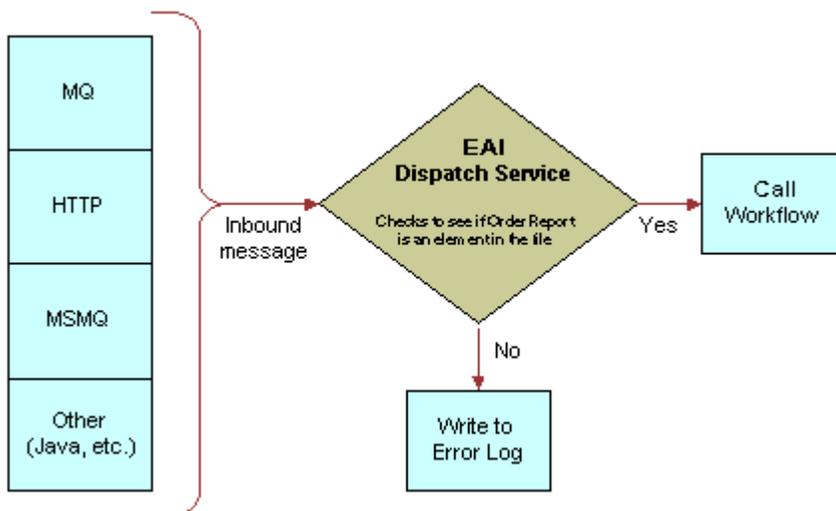
Your workflow should contain the following steps:

- o Start
- o EAI Dispatch Service
- o End

Inbound Scenario

For this scenario, you want to receive an XML document from an external system through MQ, HTTP, MSMQ, or other means and have the EAI Dispatch Service write to an error file if certain criteria are not met, as illustrated in the following figure. The scenario assumes that:

- You are only interested in the message if it contains an OrderReport element; otherwise, you want an error written to the error log.
- You know how to create a workflow.



To create this scenario

1. Create a rule set with a rule that searches the message for the OrderReport element.
2. Create a workflow that contains the following steps:
 - o Start
 - o EAI Data Mapping Engine
 - o EAI Siebel Adapter
 - o End
3. Create an EAI Dispatch Service that triggers your workflow, once the criteria in Step 1 are matched.

Outbound Scenarios Using ProcessAggregateRequest

The ProcessAggregateRequest method allows you to have multiple invocation of one or more methods in one or more business services using a single request. The following examples illustrate the use of this method to query account and employee information.

Querying the Account Integration Object

The following example shows how you can invoke multiple business services and setting arguments for each of the services. This is done using simple arguments for the services and by having the aggregate request invoke the QueryPage method of the EAI Siebel Adapter twice, with different searchspecs.

```
<?xml version="1.0" encoding="UTF-8"?>
<?Siebel-Property-Set EscapeNames="true"?>
<PropertySet>
  <AggregatedServiceRequest>
    <BusinessServiceWrapper
      BusinessServiceName="EAI Siebel Adapter"
      BusinessServiceMethod="QueryPage">
      <Argument Wrapper
        PageSize="4"
        StartRowNum="0"
        OutputIntObjectName="Sample Account" SearchSpec="[Account.Name] LIKE 'Aa*'" />
      </BusinessServiceWrapper>
    <BusinessServiceWrapper
      BusinessServiceName="EAI Siebel Adapter"
      BusinessServiceMethod="QueryPage">
      <ArgumentWrapper
        PageSize="4"
        StartRowNum="0"
        OutputIntObjectName="Sample Account" SearchSpec="[Account.Name] LIKE 'Bb*'" />
      </BusinessServiceRequest>
    </AggregatedServiceRequest>
  </PropertySet>
```

Querying the Employee Integration Object

The following example shows how you can set complex type business service method arguments. The aggregate request invokes the EAI Siebel Adapter twice, and, instead of using searchspec, uses query by example by passing in a SiebelMessage.

Note: All simple arguments are attributes of the ArgumentWrapper element, and the complex argument is a child element.

```
<?xml version="1.0" encoding="UTF-8" ?>
<?Siebel-Property-Set EscapeNames="true"?>
<PropertySet>
<AggregatedServiceRequest>
  <BusinessServiceWrapper
    BusinessServiceName="EAI Siebel Adapter"
    BusinessServiceMethod="Query">
    <ArgumentWrapper>
      <SiebelMessage
        MessageType="Integration Object"
        IntObjectName="Sample Employee"
        IntObjectFormat="Siebel Hierarchical">
        <ListOfSampleEmployee>
```

```

<Employee EMailAddr="firstname.lastname@oracle.com" />
</ListOfSampleEmployee>
</SiebelMessage>
</ArgumentWrapper>
</BusinessServiceWrapper>
<BusinessServiceWrapper
BusinessServiceName="EAI Siebel Adapter"
BusinessServiceMethod="Query">
<ArgumentWrapper>
<SiebelMessage
MessageType="Integration Object"
IntObjectName="Sample Employee"
IntObjectFormat="Siebel Hierarchical">
<ListOfSampleEmployee>
<Employee FirstName="John" LastName="Doe"/>
</ListOfSampleEmployee>
</SiebelMessage>
</ArgumentWrapper>
</BusinessServiceWrapper>
</AggregatedServiceRequest>
</PropertySet>
    
```

Examples of Search Expression Grammar

In the following example, assume that the XML document is a typical document your system receives and that you want to set some rules for the EAI Dispatch Service to use to parse this document.

```

<?xml version="1.0" encoding="UTF-8" ?>
- <cXML payloadID="3223232@ariba.acme.com" timestamp="1999-03-12T18:39:09-08:00"
xml:lang="en-US">
- <Header>
- <From>
- <Credential domain="AribaNetworkUserId">
<Identity>admin@acme.com</Identity>
</Credential>
- <Credential domain="AribaNetworkUserId" type="marketplace">
<Identity>bigadmin@marketplace.org</Identity>
</Credential>
- <Credential domain="BT">
<Identity>2323</Identity>
</Credential>
</From>
- <To>
- <Credential domain="DUNS">
<Identity>942888711</Identity>
</Credential>
</To>
- <Sender>
- <Credential domain="AribaNetworkUserId">
<Identity>admin@acme.com</Identity>
<SharedSecret>abracadabra</SharedSecret>
</Credential>
<UserAgent>Ariba.com Network V1.0</UserAgent>
</Sender>
</Header>
- <Request deploymentMode="test">
- <OrderRequest>
- <OrderRequestHeader orderID="D01234" orderDate="1999-03-12" type="new">
- <Total>
<Money currency="USD">12.34</Money>
</Total>
    
```

```
- <ShipTo>
.....
.....
```

The following table provides some valid search expression examples.

Search Expression	Description
<code>/*/Header</code>	Go to the second level and look at the type value of each property set and check whether it is of value Header.
<code>/*/*@DeploymentMode</code>	Go to the second level and look at the properties of each property set and check whether any of them has the name (not the value) of DeploymentMode.
<code>/*/*/Request@DeploymentMode</code>	Go to the third level and look at each property set for type of value Request and property of name DeploymentMode.
<code>/cXML/*/OrderRequest</code>	Search at the highest level for type with value cXML and then upon matching, find a grandchild (not child) of type with value OrderRequest.

Following are examples of invalid rules:

Rule

```
/*/*@DeploymentMode/Request/SiebelMessage
```

Description

This is not a valid rule. A search for a property value must be specified at the very end. A correct form would be the following, which will have a different result.

```
/*/Request/*@DeploymentMode
```

Rule

```
/*@PayLoadID@TimeStamp
```

Description

This also is not a valid rule. It is not possible to specify more than one property name. The correct form would use two different rules to represent this:

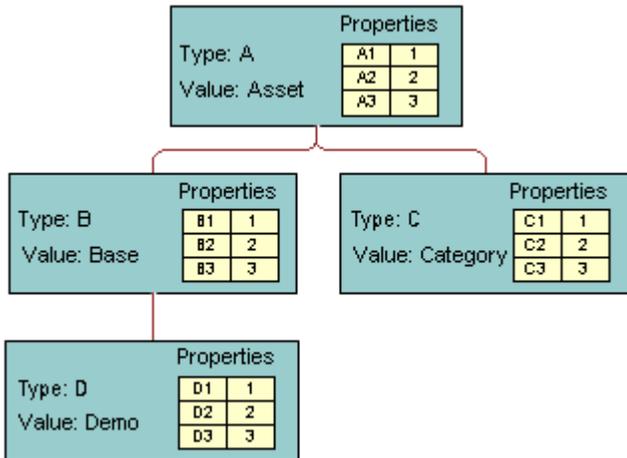
```
/*@PayLoadID
```

and

```
/*@TimeStamp
```

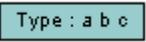
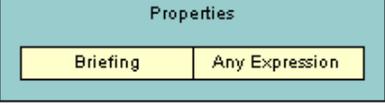
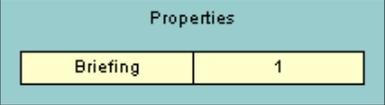
Examples of Dispatch Output Property Sets

This example shows different output property sets generated by EAI Dispatch Service based on the hierarchy input shown in the following image and certain Target and Source Expression as shown in the first table in this topic.



The following table presents the intermediate output based on the value of the Target.

Target	Source Expression	Property Name	Output Property Set
RootHierarchy	/*	N/A	
RootHierarchy	*/B	N/A	
RootHierarchy	*//*@C1	N/A	C
ChildHierarchy	/*	N/A	

Target	Source Expression	Property Name	Output Property Set
ChildHierarchy	/*/*/*D	N/A	
Type	"abc"	N/A	
Type	/*/*B	N/A	
Type	/*/*@B1	N/A	
Value	"abc"	N/A	
Property	"Any Expression"	Briefing	
Property	/*/*/*@D1	Briefing	

You can also combine different Targets to search the input message as shown on the following table.

Target	Source Expression	Property Name	Output Property										
RootHierarchy	/*	N/A	<table border="1"> <thead> <tr> <th colspan="2">Properties</th> </tr> </thead> <tbody> <tr> <td>Type: demo</td> <td>A1 this</td> </tr> <tr> <td>Value:</td> <td>A2 2</td> </tr> <tr> <td></td> <td>A3 3</td> </tr> <tr> <td></td> <td>f that</td> </tr> </tbody> </table>	Properties		Type: demo	A1 this	Value:	A2 2		A3 3		f that
Properties													
Type: demo	A1 this												
Value:	A2 2												
	A3 3												
	f that												
ChildHierarchy	/*/*/*D	N/A											
ChildHierarchy	/*/*/*@C1	N/A											
Type	"demo"	N/A											
Property	"this"	A1											
Property	"that"	f											
ChildHierarchy	/*/*/*D	N/A	<table border="1"> <thead> <tr> <th colspan="2">Properties</th> </tr> </thead> <tbody> <tr> <td>Type: demo</td> <td>A1 this</td> </tr> <tr> <td>Value:</td> <td>f that</td> </tr> </tbody> </table>	Properties		Type: demo	A1 this	Value:	f that				
Properties													
Type: demo	A1 this												
Value:	f that												
ChildHierarchy	/*/*/*@C1	N/A											
Type	"demo"	N/A											
Property	"this"	A1											
Property	"that"	f											

4 Data Mapping Using the Siebel Data Mapper

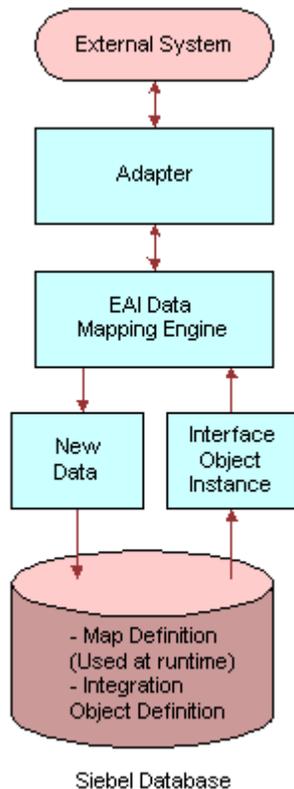
Data Mapping Using the Siebel Data Mapper

This chapter describes the process of using the Siebel Data Mapper to convert your external data to the Siebel format and your Siebel data to your external data specifications. It contains the following topics:

- *Siebel Data Mapper Overview*
- *EAI Data Mapping Engine*
- *The Siebel Data Mapper*
- *Creating Data Maps*
- *Examples of Workflow Processes*
- *About Executing Workflows*
- *EAI Data Mapping Engine Expressions*
- *Addressing Fields in Components*
- *Data Mapping Scenario*

Siebel Data Mapper Overview

The Siebel Data Mapper provides you with a declarative interface to specify maps for both inbound and outbound data transformation. The maps you set up using the Siebel Data Mapper call the EAI Data Mapping Engine to complete the data transformation. Using the Siebel Data Mapper can often reduce or even eliminate the number of scripts you need to write. The following illustrates the Siebel Data Mapper architecture.



For data mapping within Siebel, Siebel applications now support two data mapping solutions, the Siebel Data Mapper and Siebel eScript Data Mapping. The Siebel Data Mapper has a declarative interface and requires no programming skills. The Siebel eScript Data Mapping uses scripts programmed in eScript as data maps. Because the Siebel Data Mapper is based on a declarative interface, it does not have the flexibility that script-based data mapping has. Use Siebel Data Mapper for most of your integration needs, except for complex mapping situations requiring aggregation, joins, or programmatic flow control.

EAI Data Mapping Engine

To use the EAI Data Mapping Engine, you must enable the following component groups:

- Siebel Workflow
- Siebel EAI

Note: The display name of the EAI Data Transformation Engine business service in Siebel Tools is EAI Data Mapping Engine. Throughout this guide, it is referred to by both names interchangeably.

EAI Data Mapping Engine Methods

The EAI Data Mapping Engine business service has two methods: Execute and Purge.

Execute

Use the Execute method when your integration requires data transformation. Input and output arguments for the Execute method are shown in the following two tables.

Input Argument	Description
Map Name	Name of your data map.
Output Integration Object Name (Optional)	The target integration object in your map. If you use this argument you have to match it with the data map.
Siebel Message	The instance of your source integration object.
Map Arguments (Optional)	Used as an argument when you call your map from a workflow.

Property Name	Description
Name of the property	The output integration object in Siebel Message format.

Purge

This method is only for development mode. Use the Purge method to purge the database of an existing map. Use this method when you have made a change to a map and you would like to run Execute after these changes. This method does not require any input or output arguments.

Using the EAI Data Mapping Engine

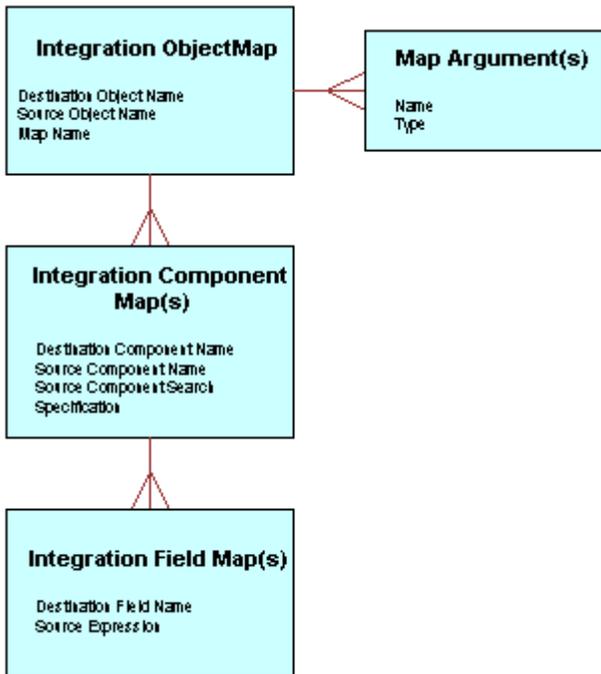
The following checklist outlines the main steps required to use the EAI Data Mapping Engine.

	Checklist
#	Create integration objects. For details, see Define Integration Objects .
#	Create data maps. For details, see Creating New Data Maps .
#	Validate data maps. For details, see Validating the Data Map .

The Siebel Data Mapper

The Siebel Data Mapper maps one integration object, source, to another integration object, target. Integration objects contain one or more integration components, which in turn contain one or more integration fields. For details on integration objects, see *Integration Platform Technologies: Siebel Enterprise Application Integration*.

The following illustrates the Siebel Data Mapping Engine architecture.



A data map defines the relationship between source and target object format. The map controls the transformation process. Transformation maps are stored in the Siebel Database as explained in the following table.

Map Type	Siebel Data Table
Integration Object maps	S_INT_OBJMAP
Integration Object Component maps	S_INT_COMPMAP
Integration Object Field maps	S_INT_FLDMAP

Integration Object Maps

An integration object map is a high-level data map specifying mapping from one integration object to another. An integration object map contains one or more integration component maps and can optionally contain integration map arguments.

Integration Map Arguments

Data maps can be parameterized using integration map arguments. Map arguments can be referenced in any expression, including the integration field map expression, source search expression, precondition expression, and postcondition expression. For example, you may want to have a field map that creates an Order Number in the target object by prefixing the Order Number in the source object with a constant.

You may want to use this map for orders coming from multiple partners and use a different prefix for each partner. To achieve this with a single data map, you can define an argument Prefix in the Integration Map Argument List, and use this argument Prefix in the field map source expression: [&Prefix]+[Order Number]. Then in the input method arguments in EAI Data Mapping Engine business service, you can specify any value for Prefix.

Integration Component Maps

Integration component maps specify how integration components in the source object get mapped to integration objects in the target object. For every occurrence of the source component in the source integration object instance, an instance of the target component is created in the target object instance. An integration component map contains one or more integration field maps. For details on integration component maps, see [Creating Integration Component Maps](#).

Integration Field Maps

Integration field maps specify how fields in the source integration object are mapped to fields in the target integration component. An integration field map target is always a field in the target component of the parent component. An integration field map source can be a constant, a reference to a map argument, a field in the source component, or other legally addressable components such as ancestors of the source component. It can also be a Siebel Query Language expression using one or more of the preceding elements.

Note: For details on integration field maps, see [Creating Integration Field Maps](#). For details on addressing fields in components other than the source component, see [Addressing Fields in Components](#). For details on Source Expression, see [Source Expression](#).

Creating Data Maps

The following checklist provides the high-level steps for creating data maps.

	Checklist
#	Define and validate integration objects and determine the required maps. For details, see Define Integration Objects
#	List components and fields within the Siebel object to use. For details, see Define Integration Objects

	Checklist
#	Create a map between the two integration objects. For details, see Creating New Data Maps .
#	Create maps between the components of the objects you mapped. For details, see Creating Integration Component Maps .
#	Create maps between individual fields within the components you mapped. For details, see Creating Integration Field Maps .
#	Validate the data maps. For details, see Validating the Data Map .

Define Integration Objects

Before you create a data map, you need to verify that valid integration objects exist for the source and the target data you want to map. For details on creating and validating integration objects, see *Integration Platform Technologies: Siebel Enterprise Application Integration*.

Determining Required Maps

The Integration Object Browser lists the existing integration object maps. Use this browser to determine which maps you need to create.

To determine which maps to create

1. Navigate to Administration - Integration, then Data Maps.
2. In the Data Maps list, query for the integration objects you want to map.

Creating New Data Maps

Once you determine what objects you need to map, use the Data Map form to create data maps. See *Define Integration Objects*.

To create a new data map

1. Navigate to Administration - Integration, then Data Maps.
2. In the Integration Object Map list, click New to create a new map.

3. Provide the necessary fields:

- **Name.** Enter a name for the map you are creating.
- **Source Object Name.** From the list of values, select the source integration object you want to create the data mapping for.
- **Target Object Name.** From the list of values, select the target integration object into which you want the data to be transferred.

Creating Maps Using Auto-Map

Once you have created your integration object map, you can use the Auto-Map button to have the Siebel application create the necessary mappings between the underlying components. The root components are always mapped by Auto-Map, whether or not they have the same name. Once the root components are mapped, the Auto-Map recursively walks through every component and their fields to map them. If the components have the same name, the Auto-Map continues to map their fields and their children components. However, if the components have different names, the Auto-Map ignores the current components, their fields, and their children components, and moves on to map the next component. In cases where only the field names are different, the Auto-Map only ignores that one field and continues with its recursive mapping.

Note: You can also use the Auto-Map on an existing mapping when you modify the integration object. The Auto-Map does not overwrite your manual mappings.

Defining Arguments for a Data Map (Optional)

After you create a data map, you can define the arguments for your map. You can then use these arguments when you call the map within workflow. To define arguments, use the Integration Map Argument list on the Integration Object Map form.

To define integration map arguments

1. Create a new record in the Integration Map Argument list.
2. Provide the following fields:
 - **Name.** Enter a name for the argument.
 - **Data Type.** From the list of values, select the Siebel Data Type for the argument.
 - **Display Name.** Enter the name that you want displayed.

Creating Integration Component Maps

Once you have defined a data map (see *Creating New Data Maps*), you need to set up the mapping between the components and the fields within the objects you have mapped. You do this using the Data Map Editor form. The Integration Object Editor list displays existing object maps and provides views in which you can define maps for components and for fields. You use the Integration Component Map view to create integration component maps.

To define integration component maps

1. Navigate to Administration - Integration, then Data Map Editor.
2. In the Integration Object Map list, select the map for which you want to define integration component maps.
3. Create a new record in the Integration Component Map list.

4. Provide the following fields.
 - o **Name.** Name of the map you are creating.
 - o **Source Component Name.** The component where you are getting the data.
 - o **Target Component Name.** The component where you want to store the data.
 - o **Source Search Specification (optional).** The search criteria based on which the records are filtered. See *Source Search Specification* for details.
 - o **Parent Component Map Name (optional).** The parent component field is used when there is a mapping to two target components that share multiple parent components. You can exclude data from one of these child objects by choosing a parent component.
 - o **Precondition (optional).** See *Preconditions* for details.
 - o **Postcondition (optional).** See *Postconditions* for details.

Creating Integration Field Maps

You define the integration field map between your source and target fields using the Integration Field Map form.

To define an integration field map

1. Create a new record in the Integration Field Map list.
2. Provide the following fields:
 - o **Target Field Name.** Name of the field in the Target Component where the value will be assigned.
 - o **Source Expression.** An expression that is used to calculate a value for the Destination Field. See *Source Expression* for details.

Validating the Data Map

Once you have created your data map, you need to validate your data map.

To validate your data map

1. Navigate to Administration - Integration, then Data Maps.
2. Select your data map.
3. Click Validate to validate your data map.
4. Take the necessary actions to fix the problems with your map or the associated integration objects.

Examples of Workflow Processes

Depending on whether you are preparing for an outbound or an inbound data exchange, you need to design different workflow processes as described in the following two procedures.

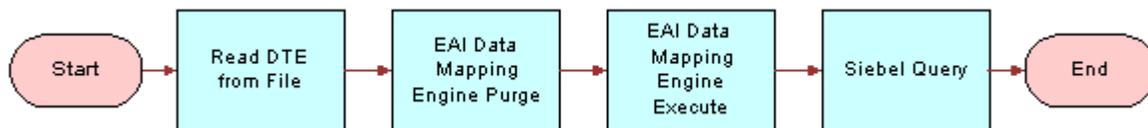
Outbound Workflow Process

To execute the map for an outbound process create a workflow process to query the database, purge the data map, execute the data map, and then write the XML into a file. The following examples illustrate integration between contact and employee business objects.

To create an outbound workflow process

1. In Siebel Tools, create a workflow process consisting of Start, End, and four business service steps. Set up each business service according to the task it needs to accomplish.

Note: Use the EAI Data Mapping Engine Purge step only in a development environment.



2. Define the following process properties:

Input Argument	Type
Employee Message	Hierarchy
IntObjName	Hierarchy
Process Instance Id	String
Error Code	String
Error Message	String
Object Id	String
Siebel Operation Object Id	String

3. The Read DTE from File step uses the EAI Siebel Adapter business service with the Query method, to query the information from the database. The business service uses the following input and output arguments.

Input Argument	Type	Value
Output Integration Object Name	literal	An Employee
Search Specification	Literal	[Employee.Last Name] LIKE "Peterson"

Input Argument	Type	Value

Property Name	Type	Output Argument
Employee Message	Output Argument	Siebel Message

Note: For more information on using the EAI Siebel Adapter, see Integration Platform Technologies: Siebel Enterprise Application Integration.

- The second business service step purges the map using the EAI Data Transformation Engine business service (display name is EAI Data Mapping Engine) with the Purge method. This step is only for development mode so that the latest map is picked for the process and should not be used in a production environment. This step does not require any input or output arguments.
- The third business service step uses the EAI Data Transformation Engine business service with the Execute method to execute the data map. The business service uses the following input and output arguments.

Input Argument	Type	Value	Property Name	Property Data Type
Map Name	Literal	Outbound DDTE Map	-	-
Output Integration Object Name	Literal	My DTE	-	-
Siebel Message	Process Property	-	Employee Message	Hierarchy

Property Name	Type	Output Argument
IntObjName	Output Argument	Siebel Message

- The Siebel Query step uses the EAI XML Write to File business service with the Write Siebel Message method to write the XML into a file. The business service uses the following input and output arguments.

Input Argument	Type	Value	Property Name	Property Data Type
File Name	Literal	c:\emp.xml	-	-
Siebel Message	Process Property	-	IntObjName	Hierarchy

Input Argument	Type	Value	Property Name	Property Data Type

The output argument for this step is optional and can be defined as follows.

Property Name	Type	Value	Output Argument
IntObjName	Output Argument	-	Siebel Message

You can use this argument to put the Siebel Message into a Hierarchy type process property.

7. Use the Workflow Process Simulator to test your workflow process.

Note: For details on creating a workflow process and using the Workflow Process Simulator to test your workflow process, see *Siebel Business Process Framework: Workflow Guide*.

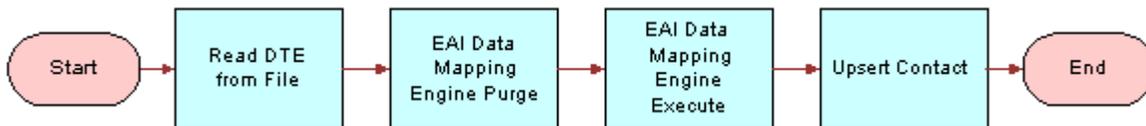
Inbound Workflow Process

To execute the map for an inbound process you need to create a workflow process to read the data from a file, purge the data map, execute the data map, and then write the XML into a file.

To create an inbound workflow process

1. In Siebel Tools, create a workflow process consisting of Start, End and four business service steps. Set up each business service according to the task it needs to accomplish.

Note: The EAI Data Mapping Engine Purge step should only be used in a development environment.



2. Define the following process properties:

Name	Data Type
Contact Message	Hierarchy
DTE Message	Hierarchy
Process Instance Id	String
Error Code	String

Name	Data Type
Error Message	String
Object Id	String
Siebel Operation Object Id	String

3. The Read DTE from File step uses the EAI XML Read from File business service with the Read Siebel Message method, to read the information from a file. The business service uses the following input and output arguments.

Input Argument	Type	Value
File Name	Literal	c:\emp.xml

Property Name	Type	Output Argument
DTE Message	Output Argument	Siebel Message

4. The second business service step purges the map using the EAI Data Transformation Engine business service (display name is EAI Data Mapping Engine) with the Purge method. This step is only for development mode so that the latest map is picked for the process and should not be used in a production environment. This step does not require any input or output arguments.
5. The third business service step uses the EAI Data Transformation Engine business service with the Execute method, to execute the data map. The business service uses the following input and output arguments.

Input Argument	Type	Value	Property Name	Property Data Type
Map Name	Literal	Inbound DDTE Map	-	-
Output Integration Object Name	Literal	A Contact	-	-
Siebel Message	Process Property	-	DTE Message	Hierarchy

Input Argument	Type	Value	Property Name	Property Data Type

Property Name	Type	Output Argument
Contact Message	Output Argument	Siebel Message

6. The Upsert Contact step uses the EAI Siebel Adapter business service with Insert or Update method to write the data into the database. This business service uses the following input argument.

Input Argument	Type	Property Name	Property Data Type
Siebel Message	Process Property	Contact Message	Hierarchy

This step does not have any output arguments.

Note: For more information on using the EAI Siebel Adapter, see *Integration Platform Technologies: Siebel Enterprise Application Integration*.

7. Use the Workflow Process Simulator to test your workflow process.

Note: For details on creating a workflow process and using the Workflow Process Simulator to test your workflow process, see *Siebel Business Process Framework: Workflow Guide*.

About Executing Workflows

Once you have designed and tested your workflows, you can run them in your production environment using Workflow Process Manager Server.

Note: For details on how to activate and execute a workflow, see *Siebel Business Process Framework: Workflow Guide*.

EAI Data Mapping Engine Expressions

The EAI Data Mapping Engine uses four categories of expressions:

- Source expressions
- Source search specifications
- Preconditions

- Postconditions

These expressions support Siebel Query Language expressions. These expressions can address fields in the source component, map arguments, and constants. In addition to fields in the source component, fields in certain other components in the source integration object can be addressed. For details, see *Addressing Fields in Components*. These expressions are just like Siebel Query Language support invocations of predefined functions and custom business services.

Note: For details on the Siebel Query Language, see *Siebel Tools Online Help*.

Source Expression

Source Expression is a required field for every integration field map. The source expression can be a literal or, based on scripting if you need to parse data or query the database for a specific value. The source expression is associated with an instance of the input integration component named in the integration component map, which is the parent of the integration field map that contains the source expression. An example of a source expression is:

```
[First Name] + " " + [Last Name]
```

This expression concatenates the First Name and the Last Name and separates them with a space to be moved into a target field such as Full Name.

Note: Only a subset of Siebel Query Language Expressions that do not require context of a business component, is supported by EAI Data Mapping Engine. You can not use the following Siebel Query Language Expressions that require context of a business component in the Source Expression: BCName(), Count(mvlink), IsPrimary(), Min(mvfield), Max(mvfield), ParentBCName(), ParentFieldValue(field_name), Sum(mvfield), GetXAVal(), GetXAValAsNum(), GetXAValAsInt(), GetXAValAsDate(), and XAIsClass().

Source Search Specification

Source Search Specification is a Boolean expression that is used to determine if a given component instance satisfies given criteria. It may only appear in an integration object map or a integration component map together with an integration component name. Defining a source search specification is optional, and if you do not define it, then it does not apply any criteria and returns True.

If a field in the current integration component has the same name as a field in a parent component, then you can only address the parent component field by using dot ('.') notation. An example of a source search specification is:

```
[Role] = "Billing"
```

The expression returns True only if the current input integration component has the value Billing in the Role field.

Note: If a source search specification is not provided, then every input integration component whose type matches the input component of the integration component map is processed.

Preconditions

You can use preconditions to make sure that a field of the input object has a certain value or otherwise terminate the process. An error is generated if the field in the input object has any other value, or no value. Preconditions are evaluated immediately before their containing integration component map is executed. If the condition is true then

the process continues. If the condition is false then the whole transformation is aborted and EAI Data Mapping Engine returns an error to the caller. An example of a precondition is:

```
[Role]=Billing Or [Role]=Shipping
```

This precondition makes sure that the field Role of the input object either has a value Billing or a value Shipping before it proceeds with the process of data transformation.

The precondition is only applied to the input components that are selected by the source search specification. The input components that fail to match the source search specification will not be checked against the precondition.

A precondition expression may address any field in the current input component, and any of its parent components. It can also address any service call parameter that has been declared as a map argument.

Note: The default value for the precondition is True. If the precondition is omitted from an integration component map then no constraint is enforced.

Postconditions

Postconditions are evaluated and applied to the newly created objects when you execute the containing integration component map. If the result of the postcondition is true then the process continues. If the result is false, the whole transformation is aborted and EAI Data Mapping Engine returns an error. Here is an example of a postcondition:

```
[Object Id]<> Or ([First Name]<> And [Last Name]<>)
```

This postcondition checks the output component for a value in the Object Id or in the First Name and the Last Name.

Note: Because there is no search specification for output components, the postcondition is applied only once for every output component instantiated because it executes its containing integration component map.

The type of the expression may be any type that can be assigned to the Destination Field type either directly or after applying standard conversions to the result of the expression.

Addressing Fields in Components

You may want to address fields in components other than the source component. This is because your target component may depend on more than one component in the source object. In such cases, you cannot use different component maps with different source components, and the same target component, because each component map creates a different instance of the target component. Data Mapping Engine expressions allow you to use the dot notation to address fields, other than the source component, in source integration object components—for example, [Component Name.Field Name].

Note: The picklist for the source expression in the Data Mapper View does not list fields in components other than the source component. Such fields should be typed in using the dot notation.

Addressing fields in other components is legal only if the cardinality of the component is less than or equal to one relative to the source component—that is, only if the component can be uniquely identified from the context of the source component without using any qualifiers other than the component name. If a field in a component that is not legally addressable is used in the source expression then it leads to a runtime error to the effect that such a field does not exist. Any component that is an ancestor of the source component in the integration object hierarchy has a

relative cardinality of 1 which means it can always be uniquely identified from the source component. Therefore, fields in ancestor components can always be legally addressed.

Sibling components can be uniquely identified from the context of the source component only if they do not occur multiple times—that is have a cardinality of less than or equal to 1. Only such siblings can be legally addressed. Therefore, it is not legal to address repeated sibling components. Components that are descendants of a sibling component can be legally addressed only if there is no repeated component in the hierarchical path from the sibling component to the component.

Further, components that are descendants of a sibling of some ancestor of a source component can be legally addressed only if there is no multiply-occurring component in the hierarchical path from the sibling-of-ancestor-of-source component to the component.

Data Mapping Scenario

The following scenarios concern an IT developer named Chris Conway, who works for a computing company, PCS Computing. One of his responsibilities is creating and maintaining the data mappings between Siebel and the other applications in use at PCS. He is assigned to create mapping between Siebel applications and the external application they need to integrate with.

Mapping Between Siebel and an External Application

Chris is in charge of integrating PCS's Siebel implementation with a custom in-house application. The purpose is to exchange customer information between the two systems.

After weighing various options, Chris decides to use the Siebel Data Mapper instead of scripts to perform the data mapping. He creates the internal integration object using the Siebel Integration Object Wizard in Siebel Tools. He also creates an external integration object using the external application's DTD.

When Chris is ready to map the two integration objects, he navigates to the Data Mapper and creates a new entry by supplying the name of the map and associating the internal integration object with the external integration object, as explained in *Creating New Data Maps*. He then uses the Map Editor form to create object, component, and field maps, as explained in *Creating Integration Component Maps*.

When he finishes creating the map, Chris creates a workflow process in Siebel Tools to define the integration flow. For one of the workflow steps, he defines an invocation of the Siebel Data Mapper. He supplies the appropriate parameters, including the name of the map, and saves his work.

5 Data Mapping Using Scripts

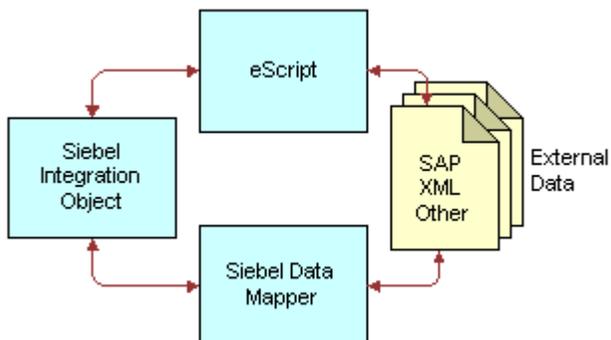
Data Mapping Using Scripts

This chapter describes the process of using the Siebel eScript Data Mapping to convert your external data to the Siebel format and your Siebel data to your external data specifications. It contains the following topics:

- *Overview*
- *EAI Data Transformation*
- *DTE Business Service Method Arguments*
- *Map Functions*
- *Data Transformation Functions*
- *Siebel Message Objects and Methods*
- *MIME Message Objects and Methods*
- *Attachments and Content Identifiers in MIME Messages*
- *XML Property Set Functions*
- *EAI Value Maps*
- *Exception Handling Considerations*
- *Sample Siebel eScript*

Overview

You can accomplish your data transformation requirements in Siebel by using the Data Transformation Function or Siebel Data Mapper, as illustrated in the following figure.



For customers who want to do data mapping within Siebel applications, Siebel applications now support two data mapping solutions—Siebel Data Mapper and Siebel eScript Data Mapping. Siebel Data Mapper has a declarative interface and requires no programming skills. Siebel eScript Data Mapping uses scripts programmed in eScript as data maps.

Data maps defined using Siebel Data Mapper are easy to maintain and upgrade. These maps also perform better than eScript Data Maps. Because Siebel Data Mapper is based on a declarative interface, it does not have the full flexibility

and power that the data mapping using eScript has. Siebel Data Mapper should suffice for most integration needs except some complex mapping situations requiring aggregation, joins, or programmatic flow control.

The following checklist outlines the main steps required to accomplish your data transformation requirements using the Data Transformation Functions.

	Checklist
#	Define integration objects in Siebel Tools—one to represent Siebel objects and another to represent external data. For details, see <i>Integration Platform Technologies: Siebel Enterprise Application Integration</i> .
#	Set up the Data Transformation Map. For details, see <i>Setting Up a Data Transformation Map</i> .
#	Write the Siebel eScript code to perform the data transformation. For details, see <i>To write a script for the DTE business service</i> .

EAI Data Transformation

The Siebel Data Transformation Functions are a framework for building data transformation maps. Data transformation maps act as import and export filters, preparing data from an external system for entry into Siebel applications and preparing data in Siebel applications for export.

Data transformation maps are created as business services using Siebel eScript. You invoke them as part of an EAI workflow process.

A data transformation map reads data from an input structure and transfers it to an output structure, transforming it along the way. The map developer creates a custom eScript function to do the transformation. The Data Transformation Functions provide a convenient way to read the input data and generate results. They also provide a framework for invoking your map functions, handling errors, and accessing other EAI resources.

Setting Up a Data Transformation Map

Using the workspace in Siebel Tools, you can create your data transformation map in a business service and you can deliver these changes into the runtime repository. You can organize your maps in many different ways. Each business service you create can contain one or more maps. You can, in fact, use several business services to organize a large number of maps into logical groups.

To define a data transformation business service in Siebel Tools

1. In Siebel Tools, create a new business service associated with a locked project.
2. Choose the CSSEAITEScriptService class for the business service.
3. Double-click the Business Services Methods folder and add the method Execute.
4. Select the Business Service Method Arg folder and add the arguments for the Execute method. For a list of arguments and their description, see *DTE Business Service Method Arguments*. The arguments to include are:

- o MapName
- o An input argument. Select one of SiebelMessage, XMLHierarchy, or MIMEHierarchy as the argument name, based on the type of input.
- o An output argument. Required if the output object is a different type than the input argument. Select one of SiebelMessage, XMLHierarchy, or MIMEHierarchy as the argument name.

Note: If the input and output types are the same then the same argument entry is used for both.

- o OutputType
- o InputType(Optional). This is required only when passing the business service input property set to the map function without interpretation. This is done by specifying the InputType as ServiceArguments.

Note: Most transform maps use SiebelMessage for both the input and output arguments. This is for mapping one integration object to another. For details, see *DTE Business Service Method Arguments*.

Once you have created the business service you need to write the Siebel eScript code to perform the data transformation.

To write a script for the DTE business service

1. In Siebel Tools, select the business service you want to contain the transformation map.
2. Right-click, then choose Edit Server Scripts
3. Choose eScripts as the scripting language if you are prompted to select a scripting language.
4. In the (declarations) procedure of the (general) object, add the line:

```
#include "eaisiebel.js"
```

5. In the Service_PreInvokeMethod function of the service, change the function to the following:

```
function Service_PreInvokeMethod (MethodName, Inputs, Outputs)
{
    return EAIExecuteMap (MethodName, Inputs, Outputs);
}
```

Your data transformation map is run as a business service invoked from a workflow process. Business service scripts have a standard entry point, Service_PreInvokeMethod. Although the script environment provides you with a boilerplate function by this name, you need to modify it, as described in the preceding steps, to include the call to the EAIExecuteMap function.

- a. The MethodName must be Execute and is used by Siebel Workflow. The name of your function is the name you supply for the MapName argument to the Execute method.
- b. Inputs is the input message from workflow containing service arguments—for example, MapName and Output Integration Object Name—and the integration message to be transformed. Outputs is the argument used to return data—for example, Siebel Message. MapName specifies the map function to be executed and must be the name of one of the functions you defined in the business service.

For examples of DTE business services, select the Business Service object in Siebel Tools, and then query for CSSEAIIDTEScriptService in the Class field in the Object List Editor.

DTE Business Service Method Arguments

The following information presents the arguments for the Execute method of the DTE business services.

Name & Display Name	Data Type	Type	Optional	Storage Type	PickField	PickList
MapName Map Name	String	Input	No	Property	-	-
InputType Input Type	String	Input	No	Property	Value	EAI Message Type PickList
OutputType Output Type	String	Input	No	Property	Value	EAI Message Type PickList
SiebelMessage Siebel Message	Hierarchy	Input/Output	Yes	Hierarchy	-	-
MIMEHierarchy MIME Hierarchy	Hierarchy	Input/Output	Yes	Hierarchy	-	-
XMLHierarchy XML Hierarchy	Hierarchy	Input/Output	Yes	Hierarchy	-	-

You can set the following arguments in Siebel Tools:

- **MapName.** The name of the eScript function to call to perform the transformation.
- **InputType.** The type of input object to pass to the transformation function. The value will be one of SiebelMessage, MIMEHierarchy, XMLHierarchy, or ServiceArguments. This argument is required only when you use ServiceArguments as the value. When ServiceArguments is used the business service, PropertySet is passed to the map function without interpretation.
- **OutputType.** The type of the output object to pass to the transformation function. The types are the same as the ones for Input Type.
- **SiebelMessage.** You use this argument when the input or output object or both are SiebelMessage. SiebelMessage is used when converting to or from an integration object, and is the correct choice when mapping one integration object to another. Your map function is passed two objects of type CSSEAllntMsgIn and CSSEAllntMsgOut. These objects are for the SiebelMessage that is the input to the transformation and the SiebelMessage that is produced by the transformation, respectively.
- **MIMEHierarchy.** You use this argument if either the input or output object or both are MIMEHierarchy. MIMEHierarchy is used when converting to or from MIME Hierarchy objects. Your map function is passed two object types; CSSEAllntMimeMsgIn for the MIMEHierarchy that is the input to the transformation and

CSSEAIMimeMsgOut for the MIMEHierarchy that is produced by the transformation. MIME Hierarchy objects are defined by the EAI MIME Doc Converter business service. For details on the EAI MIME Doc Converter, see *Integration Platform Technologies: Siebel Enterprise Application Integration*.

- **XMLHierarchy.** You use this argument if either the input or output object or both are XMLHierarchy. XMLHierarchy is used when converting to or from XML Hierarchy objects. Your map function is passed an object of type XML Property Set for both input and output XMLHierarchy. XML Hierarchy objects are defined by the XML Hierarchy Converter business service. For details on XML Hierarchy Converter, see *XML Reference: Siebel Enterprise Application Integration*.

Map Functions

A map function has the following signature:

```
function MapFnName (objectIn, objectOut)
```

The function name signified by MapFnName is the name of your transformation function. It is the value passed as the MapName argument to the business service. The Input Type and Output Type business service arguments determine the types of the objectIn and objectOut arguments and default to the type Integration Message. You should name these arguments according to type. For example, to use the default values, you would specify a function that transforms one integration object to another as:

```
function MapFnName (intMsgIn, intMsgOut)
```

If you define a function that transforms an XML property set to an integration object, you might specify it as:

```
function MapFnName (xmlPropSetIn, intMsgOut)
```

The arguments to these functions are contained within the input and output arguments to the business service's Service_PreInvokeMethod function. The EAIExecuteMap function—called by Service_PreInvokeMethod—interprets the arguments and passes them to MapFnName. MapFnName reads from the input object and writes to the output object using the appropriate API for each type of object.

If you define a function to access input integration object, you might specify it as:

```
function myMapFn (ObjectIn, ObjectOut) {  
  inIntObj = ObjectIn.GetIntObj(); //Get Integration Object  
  //Iterate over all Integration Object Instances  
  while (inIntObj.NextInstance()) {  
    //Get the Primary Component which is called "Order Entry - Orders"  
    primaryIntComp = inIntObj.GetPrimaryIntComp("Order Entry - Orders");  
    //Iterate over all instances of Primary Component  
    while (primaryIntComp.NextRecord()) {  
      OrderId = primaryIntComp.GetFieldValue ("Id");  
      //Get component "Order Entry - Line Items" which is child of "Order Entry - Orders"  
      comp = primaryIntComp.GetIntComp ("Order Entry - Line Items");  
      //Process component similar to primary component  
      while (comp.NextRecord()) {  
        OrderItemId = comp. GetFieldValue ("Id");  
      }  
    }  
  }  
}
```

And to define a function to create output integration object, you might specify it as:

```
function myMapFn (ObjectIn, ObjectOut) {  
  outIntObj = ObjectOut.CreateIntObj("Sample Order");  
  while (Need new integration object instances) {  
    outIntObj.NewInstance();  
    //Create Primary Component which is called "Order Entry - Orders"  
    primaryIntComp = outIntObj.CreatePrimaryIntComp("Order Entry - Orders");  
    while (Need new instances of primary int component) {  
      }  
    }  
}
```

```
primaryIntComp.NewRecord();  
primaryIntComp.SetFieldValue ("Id", OrdertemId);  
//Create component Order Item which is child of Order  
comp = primaryIntComp.CreateIntComp ("Order Entry - Order Items");  
//Process component similar to primary component  
while (need new instances of component) {  
  comp.NewRecord();  
  comp.SetFieldValue ("Id", OrdertemId);  
}
```

EAIExecuteMap() Method

This method executes a user-defined data transformation function. The following information presents the parameters for this method.

Syntax

EAIExecuteMap(methodName, inputPropSet, outputPropSet)

Parameter	Description
methodName	The business service method should be Execute.
inputPropSet	Input message and service arguments.
outputPropSet	Output message and service arguments.

Returns

CancelOperation or ContinueOperation. The Service_PreInvokeMethod function should return the value returned by the EAIExecuteMap.

Usage

See *Setting Up a Data Transformation Map*.

Data Transformation Functions

The data transformation API consists of global functions and classes that represent the different parts of input and output data. The data transformation functions are implemented as Siebel eScript. You must use Siebel eScript to create your data transformation maps.

Three different high-level data types are supported:

- **Siebel Messages.** See *Siebel Message Objects and Methods*.
- **MIME Messages.** See *MIME Message Objects and Methods*.
- **XML Property Sets.** See *XML Property Set Functions*.

The data type is determined by the `InputType` and `OutputType` arguments, as described in *DTE Business Service Method Arguments*.

Siebel Messages are the most common data type, and they are represented at the highest level by an Integration Message message. See *Siebel Message Objects and Methods*.

It is also possible to operate directly on the business service input and output property sets. This is accomplished by specifying the `InputType` or `OutputType` as `ServiceArguments`. In this case the business service property set arguments are passed directly to the map function. The standard property set functions can be used to access them.

Siebel Message Objects and Methods

A Siebel Message is a message containing the data of individual integration object instances. It is hierarchically structured and composed of several different types of objects.

The data transform API uses several different eScript classes to represent a Siebel Message:

- **An integration message.** This represents the high-level message container. See *Integration Message Objects*.
- **An integration object.** See *Integration Object Objects*.
- **A primary integration component.** See *Primary Integration Component Objects*.
- **Integration components.** See *Integration Component Objects*.

Each of these parts of a Siebel Message has two classes: one for input and one for output. Each class provides methods for specific purposes.

Integration Message Objects

The integration message is the high-level piece of a message. The workflow process passes the integration message to the Data Mapping Engine as input. The Data Mapping Engine returns another message as output. The integration message object provides access to workflow arguments, integration message arguments, and the integration object that is contained in the message.

The following integration message objects are provided:

- `CSSEAllIntMsgIn`
- `CSSEAllIntMsgOut`

`CSSEAllIntMsgIn`

This object represents an integration message that is open for reading. The object provides `GetArgument` and `GetIntObj` methods.

`GetArgument()` Method

This method gets the value of a business service argument. For example, this could get the name of a map function in the business service. The following information presents the parameters for this method.

Syntax

GetArgument(name [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
name	The name of a business service argument.
defaultIfNull	Returned if a service argument of the specified name does not exist.
defaultIfEmpty	Returned if the service argument is set to an empty string.

Returns

String or null.

Usage

Use this method to get the value of an argument passed to the business service. For example, if the MapName argument passed to the business service is MapExtOrderToOrder, the call:

```
intMsgIn.GetArgument("MapName");
```

returns the name of the map, MapExtOrderToOrder, passed to the business service.

If the named argument does not exist, `null` is returned. If the named argument exists but the value is the empty string, the empty string is returned. You can use the `defaultIfNull` and `defaultIfEmpty` optional arguments to change this behavior.

The arguments `defaultIfNull` and `defaultIfEmpty` are optional; however, if you specify `defaultIfEmpty`, you must also specify the `defaultIfNull` argument.

GetIntObj() Method

This method returns an instance of the integration object and opens it for reading. The following table presents the parameter for this method.

Syntax

GetIntObj(name)

Parameter	Description
name	The name of an integration object in the active integration message.

Returns

CSSEAllIntObjIn Integration Object

Usage

An integration object instance is always returned even if the integration object does not exist. Call the returned object's `Exist` method to test for this before calling other methods on the object. An error is raised if an integration object is present but the name is not correct.

Note: Currently an integration message can contain only one integration object.

GetAttachmentCount() Method

This method returns the number of attachments in the input integration message.

Syntax

`GetAttachmentcount()`

Returns

The number of attachments in the input integration message.

GetAttachment() Method

This method returns the attachment specified by the index. The following table presents the parameter for this method.

Syntax

`GetAttachment(index)`

Parameter	Description
index	The index of the attachment to return.

Returns

The attachment (a `PropertySet`) specified by the index. The index is zero based. Returns null if index is out of bounds.

GetAttachmentByCID() Method

This method retrieves an attachment based on the Content Identifier (CID).The following table presents the parameter for this method.

Syntax

`GetAttachmentByCID(cid)`

Parameter	Description
cid	The Content Identifier of the attachment.

Returns

The attachment (a PropertySet) specified by the CID. Returns null if there is no attachment with the specified CID.

CSSEAllIntMsgOut

This object represents an output integration message that is open for writing. The object provides CreateIntObj and SetArgument methods:

CreateIntObj() Method

This method creates a new integration object. The following information presents the parameter for this method.

Syntax

CreateIntObj(name)

Parameter	Description
name	Creates a new integration object and adds it to the integration message.

Returns

CSSEAllIntObjOut Output Integration Object

Usage

An integration message can contain only one integration object, so multiple calls to this method on one integration message raises an error. The name must agree with the business service argument OutputIntObjName, if that argument is passed to the service.

SetArgument() Method

This method sets the value of a business service argument. The following table presents the parameters for this method.

Syntax

SetArgument(name, value)

Parameter	Description
name	The name of an argument in the active business service.
value	The string value corresponding to the argument named by the <i>name</i> parameter.

Returns

Not applicable

Usage

You can call the `SetArgument` method to establish the value of a given output argument for the business service method invocation.

SetAttachmentSource() Method

This method establishes the source object to copy attachment objects from. The source object must be a `CSSEAllntMsgIn`, `CSSEAIMimeMsgIn`, or other object implementing the `GetAttachmentByCID` method. The following table presents the parameter for this method.

Syntax

`SetAttachmentSource(source)`

Parameter	Description
source	The attachment source.

CopyAttachment() Method

This method copies an attachment from the attachment source to the output integration object. The attachment is referenced by the MIME Content Identifier (CID). The attachment source must be established by calling `CSSEAllntMsgOut.SetAttachmentSource` before calling this method. The following table presents the parameter for this method.

Syntax

`CopyAttachment(cid)`

Parameter	Description
cid	MIME content identifier.

Returns

The attachment copy is returned as a property set. This method returns null if the attachment source does not contain an attachment with the specified CID.

Integration Object Objects

The integration object contains one or more integration components. The following integration object objects are provided:

- CSSEAllIntObjIn
- CSSEAllIntObjOut

CSSEAllIntObjIn

This object represents an input integration object, open for reading, that is contained in the integration message. The integration object has a name and contains zero or more instances of actual integration objects. Integration object instances are accessed one at a time, similar to accessing database records. Each instance has a primary integration component that contains data and every subordinate integration components. This object provides the Exists, FirstInstance, GetPrimaryIntComp, and NextInstance methods.

Exists() Method

This method checks to see if the integration object is actually present in the input data. It takes no parameters.

Syntax

Exists()

Returns

Boolean

Usage

Call Exists after retrieving the integration object from the integration message. If the integration object was found and is open for reading, the Exists method returns true.

FirstInstance() Method

This method moves to the first integration object instance and sets it as the active instance.

Syntax

FirstInstance()

Returns

Boolean

Usage

The FirstInstance method returns true if the instance exists, false otherwise.

GetPrimaryIntComp() Method

This method returns the primary integration component of the active instance of the integration object. The following information presents the parameter for this method.

Syntax

GetPrimaryIntComp(name)

Parameter	Description
name	The name of a primary integration component in the active integration object instance.

Returns

CSSEAIPrimaryIntCompIn Input Primary Integration Component

Usage

Gets the primary integration component of the active instance of the integration object and opens it for input.

This method always returns an input primary integration component object, even if the component does not exist. Call the Exists method on the returned object to test for this condition. If there is no active instance, a call to this method raises an error.

NextInstance() Method

This method moves a pointer to the next logical integration object instance in the active integration message.

Syntax

NextInstance()

Returns

Boolean

Usage

Moves to the next integration object instance and makes it the active instance. This method returns true if the instance exists, or false if there are no more instances. If neither the NextInstance or the FirstInstance method has been called previously, the NextInstance method moves to the first instance in the message.

CSSEAIIntObjOut

This object represents an output integration object, open for writing, that is contained in the integration message. It provides CreatePrimaryIntComp and NewInstance methods as an interface to the output integration object.

CreatePrimaryIntComp() Method

This method creates a new primary integration component. The following information presents the parameter for this method.

Syntax

CreatePrimaryIntComp(name)

Parameter	Description
name	Assigned as the name of the Primary Integration Component.

Returns

CSSEAIPrimaryIntCompOut Primary Integration Component, open for output

Usage

Use the Exists method to test for existence of the integration object instance, then create a new integration object instance and set it as the active instance, using the NewInstance method. You must perform these tasks before calling the CreatePrimaryIntComp() method.

NewInstance() Method

This method creates a new instance of an integration object and makes it the active instance.

Syntax

NewInstance()

Returns

Not applicable

Primary Integration Component Objects

A primary integration component represents the integration component contained within an integration object instance. It has a name and contains records with data from actual integration components. Each record may have fields and subordinate integration components.

The following primary integration component objects are provided:

- CSSEAIPrimaryIntCompIn
- CSSEAIPrimaryIntCompOut

CSSEAIPrimaryIntCompln

This object represents the input primary integration component, open for reading. Your data transformation maps can use this object's methods to traverse integration components. This object provides Exists, FirstRecord, GetFieldValue, GetIntComp, and NextRecord methods:

Exists() Method

This method checks to see if the primary integration component is actually present in the input data. It takes no parameters.

Syntax

Exists()

Returns

Boolean

Usage

Call Exists after retrieving the primary integration component with the CSSEAIIntObjIn.GetPrimaryIntComp method, and before invoking the primary integration component's other methods.

If the primary integration component was found and is open for reading, the Exists method returns true.

FirstRecord() Method

This method moves a pointer to the first component record in the primary integration component.

Syntax

FirstRecord()

Returns

Boolean

Usage

Moves to the first integration component record and sets it as the active record. This method returns true if the record exists, false if the integration component has no records.

GetFieldValue() Method

This method returns the value of the primary integration component field from the active record. The following information presents the parameters for this method.

Syntax

GetFieldValue(name [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
name	The name of a primary integration component field.
defaultIfNull	Optional. Sets the default value if the field does not exist.
defaultIfEmpty	Optional. Sets the default value if the field is set to an empty string.

Returns

String or null

Usage

A null value is returned if the active record does not contain the field. Otherwise, a string containing the value in the field is returned. If there is no active record, this method raises an error.

If the named argument does not exist, null is returned. If the named argument exists but the value is the empty string, the empty string is returned. You can use the `defaultIfNull` and `defaultIfEmpty` optional arguments to change this behavior.

The arguments `defaultIfNull` and `defaultIfEmpty` are optional; however, if you specify `defaultIfEmpty`, you must also specify the `defaultIfNull` argument.

GetIntComp() Method

This method returns the named integration component from the active record and opens it for input. The following table presents the parameter for this method.

Syntax

`GetIntComp(name)`

Parameter	Description
name	The name of an integration component in the active record.

Returns

CSSEAllntCompln Input Integration Component

Usage

This method always returns an input integration component object, even if the component does not exist. Call the `Exists` method on the returned object to test for this condition. If there is no active record, a call to this method raises an error.

NextRecord() Method

This method moves a pointer to the next logical record in the active integration component.

Syntax

NextRecord()

Returns

Boolean

Usage

Moves to the next record and makes it the active record. Returns true if the record exists, or false if there are no more records. Moves to the first record if neither the NextRecord method nor the FirstRecord method has been called previously.

CSSEAIPrimaryIntCompOut

This object represents the output primary integration component. You can use the object's methods to create output integration components and records and to copy input data records to output data records. This object provides CopyFieldValue, CreateIntComp, NewRecord, SetCopySource, and SetFieldValue methods.

CopyFieldValue() Method

This method sets the value of a field in the active record to the value of a field in the current source record. The following information presents the parameters for this method.

Syntax

CopyFieldValue(targetName, sourceName [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
targetName	Name of the field to set in the output integration component.
sourceName	Name of the field to retrieve from the input integration component.
defaultIfNull	Optional value that specifies what should be inserted into the target, if the source field does not exist.
defaultIfEmpty	Optional value that specifies what to use as a source value if the source field is empty.

Returns

Not applicable

Usage

Use this method to copy a field from an input integration component to the output primary integration component. You could achieve the same results by calling the GetFieldValue method on the input component and the SetFieldValue on the output component; however, using CopyFieldValue is easier.

You must call the `SetCopySource` method first to specify the source integration component. `CopyFieldValue` uses the active records of the input and output components of the active integration component.

If the integration component is not set with the `SetCopySource` method first, a call to the `CopyFieldValue` method raises an error. An error also occurs if either input or output component does not have an active record.

If you set the copy source using the following statement:

```
outIntComp.SetCopySource (inIntComp);
```

the following two statements are equivalent:

```
outIntComp.SetFieldValue("Fld-A", inIntComp.GetFieldValue("X"));  
outIntComp.CopyFieldValue("Fld-A", "X");
```

Using the second convention is convenient if you are copying many fields between the same components.

CreateIntComp() Method

This method creates a new integration component. The following table presents the parameters for this method.

Syntax

`CreateIntComp(name [, createNow])`

Parameter	Description
name	The name of the new integration component.
createNow	Defaults to true. This is an optional parameter. By default, the underlying data object is created in the output data object at the time this method is called. To change this behavior, specify the optional <code>createNow</code> argument as false. If you specify <code>createNow</code> as false, the underlying data object is not created until you make the first <code>NewRecord</code> call on the newly created integration component.

Returns

`CSSEAllIntCompOut`. Output Integration Component

Usage

Use this method to create a new integration component, open it for writing, and add it to the active record of the integration component.

Note: This method raises an error if you call it without an active integration component record. Use the `NewRecord` method to create a new record and set the active record.

NewRecord() Method

This method creates a new record in a primary integration component.

Syntax

`NewRecord()`

Returns

Not applicable

Usage

This method adds a new primary integration component record and makes it the active record.

SetCopySource() Method

This method establishes the integration component from which a field value will be copied. The following table presents the parameter for this method.

Syntax

SetCopySource(IntComp)

Parameter	Description
IntComp	The integration component object—either CSSEAIPrimaryIntCompln or CSSEAllntCompln.

Returns

Not applicable

Usage

Call this method before a call to the CopyFieldValue method.

SetFieldValue() Method

This method sets the value of the named field in the active integration component record. The following table presents the parameters for this method.

Syntax

SetFieldValue(name, value)

Parameter	Description
name	The name of a field in the active record of the primary integration component.
value	The string value to be put into the field given in the name parameter.

Returns

Not applicable

Usage

Both the name and value arguments should be strings.

The field is not set if the value is `null`. This method provides no return value.

This method raises an error if called while there is no active record.

Note: Siebel eScript automatically converts most types to strings as necessary.

Integration Component Objects

An integration component object represents integration components. The following integration component objects are provided:

- CSSEAllIntCompIn
- CSSEAllIntCompOut

CSSEAllIntCompIn

This object represents the input integration component, open for reading. You can use the object's methods to traverse actual integration components and to retrieve data from those integration components. This object provides `Exists`, `FirstRecord`, `GetFieldValue`, `GetIntComp`, and `NextRecord` methods.

Exists() Method

This method checks to see if the integration component is actually present in the input data. It takes no parameters.

Syntax

`Exists()`

Returns

Boolean

Usage

Call `Exists` after retrieving the integration component from its parent object using the `GetIntComp` method, and before invoking the integration component's other methods.

If the integration component is found and is open for reading, the `Exists` method returns `true`.

FirstRecord() Method

This method moves a pointer to the first component record in the integration component.

Syntax

`FirstRecord()`

Returns

Boolean

Usage

Moves to the first integration component record and sets it as the active record. This method returns true if the record exists, false if the integration component has no records.

GetFieldValue() Method

This method returns the value of the integration component field from the active record. The following information presents the parameters for this method.

Syntax

GetFieldValue(name [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
name	The name of an integration component field.
defaultIfNull	Optional. Value to return if the field does not exist.
defaultIfEmpty	Optional. Value to return if the field is set to an empty string.

Returns

String or `null`

Usage

A `null` value is returned if the active record does not contain the field. Otherwise, a string containing the value in the field is returned. If there is no active record, this method raises an error.

If the named argument does not exist, `null` is returned. If the named argument exists but the value is the empty string, the empty string is returned. You can use the `defaultIfNull` and `defaultIfEmpty` arguments to change this behavior.

Note: The arguments `defaultIfNull` and `defaultIfEmpty` are optional. However, if you specify `defaultIfEmpty`, you must also specify the `defaultIfNull` argument.

GetIntComp() Method

This method returns the integration component from the active record and opens it for input. The following table presents the parameter for this method.

Syntax

GetIntComp(name)

Parameter	Description
name	The name of an integration component in the active record.

Returns

CSSEAllntCompIn Input Integration Component

Usage

This method always returns an input integration component object, even if the component does not exist. Call the Exists method on the returned object to test for this condition.

Note: If there is no active record, a call to this method raises an error.

NextRecord() Method

This method moves a pointer to the next logical record in the active integration component.

Syntax

NextRecord()

Returns

Boolean

Usage

Moves to the next record and makes it the active record. Returns true if the record exists, or false if there are no more records. Moves to the first record if neither the NextRecord method nor the FirstRecord method has been called previously.

CSSEAllntCompOut

This object represents the output integration object, open for writing. You can use this object's methods to create new output integration components and to copy or set actual data in the records of the integration components. This object provides CopyFieldValue, CreateIntComp, NewRecord, SetCopySource, and SetFieldValue methods.

CopyFieldValue() Method

This method sets the value of a field in the active record to the value of a field in the current source record. The following information presents the parameters for this method.

Syntax

CopyFieldValue(targetName, sourceName [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
targetName	Name of the field to set in the output integration component.
sourceName	Name of the field to retrieve from the input integration component.
defaultIfNull	Optional value that specifies what should be inserted into the target, if the source field does not exist.
defaultIfEmpty	Optional value that specifies what to use as a source value if the source field is empty.

Returns

Not applicable

Usage

Use this method to copy a field from an input integration component to the output integration component. You could achieve the same results by calling the `GetFieldValue` method on the input component and the `SetFieldValue` on the output component; however, using `CopyFieldValue` is easier.

You must call the `SetCopySource` method first to specify the source integration component. `CopyFieldValue` uses the active records of the input and output components of the active integration component.

If the integration component is not set with the `SetCopySource` method first, a call to the `CopyFieldValue` method raises an error. An error also occurs if either input or output component does not have an active record.

If you set the copy source using the following statement:

```
outIntComp.SetCopySource(inIntComp);
```

the following two statements are equivalent:

```
outIntComp.SetFieldValue("Fld-A", inIntComp.GetFieldValue("X"));
outIntComp.CopyFieldValue("Fld-A", "X");
```

Using the second convention is convenient if you are copying many fields between the same components.

CreateIntComp() Method

This method creates a new integration component. The following table presents the parameters for this method.

Syntax

CreateIntComp(name [, createNow])

Parameter	Description
name	The name of the new integration component.
createNow	Defaults to true. This is an optional parameter. By default, the underlying data object is created in the output data object at the time this method is called. To change this behavior, specify the optional <code>createNow</code> argument as false. If you specify <code>createNow</code> as false, the underlying data object is not created until you make the first <code>NewRecord</code> call on the newly created integration component.

Parameter	Description

Returns

CSSEAllntCompOut. Output Integration Component

Usage

Use this method to create a new integration component, open it for writing, and add it to the active record of the integration component.

This method raises an error if you call it without an active integration component record. Use the NewRecord method to create a new record and set the active record.

SetCopySource() Method

This method establishes the integration component from which a field value will be copied. The following table presents the parameter for this method.

Syntax

SetCopySource(IntComp)

Parameter	Description
IntComp	The integration component object—either CSSEAIPrimaryIntCompln or CSSEAllntCompln.

Returns

Not applicable

Usage

Call this method before calling the CopyFieldValue method.

SetFieldValue() Method

This method sets the value of the named field in the active integration component record. The following table presents the parameters for this method.

Syntax

SetFieldValue(name, value)

Parameter	Description
name	The name of a field in the active record of the integration component.
value	The string value to be put into the field given in the name parameter.

Parameter	Description

Returns

Not applicable

Usage

Both the name and value arguments should be strings.

The field is not set if the value is `null`. This method provides no return value.

This method raises an error if called while there is no active record.

Note: Siebel eScript automatically converts most types to strings as necessary.

MIME Message Objects and Methods

Siebel EAI represents MIME documents using a property set format. This is the format used by the EAI MIME Doc Converter Business Service. The objects and methods described here provide access to this property set format, and are intended for use in conjunction with transforming pieces of the MIME message to and from Siebel Integration Messages.

Note: The EAI MIME Hierarchy Converter Business Service is the preferred method of converting between the property set representation Siebel Messages.

The following MIME message objects are provided:

- CSSEAIMimeMsgIn
- CSSEAIMimeMsgOut

CSSEAIMimeMsgIn

This object represents an input MIME Message, open for reading. The MIME message is in the property set format generated by the EAI MIME Doc Converter. The object consists of a series of MIME parts forming the different pieces of the message.

This object provides `GetArgument`, `GetPartCount`, `GetPart`, `GetPartByCID`, `GetAttachmentByCID`, and `GetXMLRootPart` methods.

GetArgument() Method

This method gets the value of a business service argument. For example, this could get the name of a map function in the business service. The following information presents the parameters for this method.

Syntax

GetArgument(name [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
name	The name of a business service argument.
defaultIfNull	Returned if a service argument of the specified name does not exist.
defaultIfEmpty	Returned if the service argument is set to an empty string.

Returns

String or `null`

Usage

Use this method to get the value of an argument passed to the business service. For example, if the `MapName` argument passed to the business service is `MapExtOrderToOrder`, the call:

```
intMsgIn.GetArgument("MapName");
```

returns the name of the map, `MapExtOrderToOrder`, passed to the business service.

If the named argument does not exist, `null` is returned. If the named argument exists but the value is the empty string, the empty string is returned. You can use the `defaultIfNull` and `defaultIfEmpty` optional arguments to change this behavior.

The arguments `defaultIfNull` and `defaultIfEmpty` are optional; however, if you specify `defaultIfEmpty`, you must also specify the `defaultIfNull` argument.

GetPartCount() Method

This method returns the number of parts in the MIME message. The following table presents the parameter for this method.

Syntax

GetPartCount()

Returns

This method returns the number of parts in the MIME message.

GetPart() Method

Syntax

GetPart(index)

Parameter	Description
index	Index of the MIME part to return.

Returns

Property set. Returns the part, a property set, specified by the index. The index is zero based. Returns null if the index is out of bounds.

GetPartByCID() Method

Retrieve a MIME part based on the MIME Content Identifier (CID).The following table presents the parameter for this method.

Syntax

GetPartByCID(cid)

Parameter	Description
cid	MIME Content Identifier to retrieve.

Returns

Returns null if there is no part with the specified CID.

GetAttachmentByCID() Method

The same functionality as CSSEAIMimeMsgIn.GetPartByCID. Supports using a CSSEAIMimeMsgIn as an attachment source for copying attachments to output objects. The following table presents the parameter for this method.

Syntax

GetAttachmentByCID(cid)

Parameter	Description
cid	MIME Content Identifier to retrieve.

Returns

The attachment (a property set) specified by the CID. Returns null if there is no attachment with the specified CID.

GetXMLRootPart() Method

Finds the first MIME part that is an XML message in property set format and returns the root element of the XML document. The XML message must be in property set format as produced by the XML Hierarchy Converter Business Service. An error is raised if the XML message is not found. The method is intended for use with MIME messages that consist of an XML message and a series of related attachments. The property set returned is consistent with

what XPSGetRootElement returns, and can be accessed with the XML Property Set functions. See *XML Property Set Functions*.

Syntax

GetXMLRootPart()

Returns

MIME body part representing an XML document.

CSSEAIMimeMsgOut

This object represents an output MIME message, open for writing. This object provides SetArgument, CreateXMLPart, SetAttachmentSource, and CopyAttachment methods:

SetArgument() Method

This method sets the value of a business service argument. The following information presents the parameters for this method.

Syntax

SetArgument(name, value)

Parameter	Description
name	The name of an argument in the active business service.
value	The string value corresponding to the argument named by the <i>name</i> parameter.

Returns

Not applicable

Usage

You can call the SetArgument method to establish the value of a given output argument for the business service method invocation.

CreateXMLPart() Method

This method is similar to XPSCreateRootElement. See *XPSCreateRootElement()*. CreateXMLPart() Method creates an XML MIME part and adds it to the MIME document. The property set representing the XML root element is returned. The property set returned can be populated using the XML Property Set functions. See *XML Property Set Functions*. The following table presents the parameter for this method.

Syntax

CreateXMLPart(xmlRootTagName)

Parameter	Description
xmlRootTagName	The name you want to supply as the root element name in the XML document.

Returns

Property set

SetAttachmentSource() Method

This method establishes the source object from which to copy attachment objects. The source object must be a CSSEAIIntMsgIn, CSSEAIMimeMsgIn, or other object implementing the GetAttachmentByCID method. The following table presents the parameter for this method.

Syntax

SetAttachmentSource(source)

Parameter	Description
source	The attachment source.

CopyAttachment() Method

This method copies an attachment from the attachment source to the output MIME message object. The attachment is referenced by the MIME Content Identifier (CID). The attachment copy, a property set, is returned. The attachment source must be established by calling CSSEAIMimeMsgOut.SetAttachmentSource before calling this method. The following table presents the parameter for this method.

Syntax

CopyAttachment(cid)

Parameter	Description
cid	MIME Content Identifier of the attachment to copy.

Returns

Property set. This method returns null if the attachment source does not contain an attachment with the specified CID.

Attachments and Content Identifiers in MIME Messages

A MIME message contains one or more parts, each representing a separate piece of the message. One common use of multipart MIME messages is to include attachments with a message.

Note: All the examples have to be typed single-spaced and without word wrap.

Each MIME body part has an optional Content Identifier (CID) used to identify it. The Content Id is part of the MIME part header, for example:

```
--unique_boundary_123
Content-Type : image/jpeg
Content-ID : <001110.102215@abc.com>
```

Then the CID is 001110.102215@abc.com. The CID is usually referenced from another part of the MIME message. A common scheme is to use an XML document as the main part of the MIME message, and use Content Ids to reference the other attachments in the message. The following is an example of a MIME message with attachment.

```
MIME-Version: 1.0
Content-Type: multipart/related;
  boundary="unique_boundary_123";
  type="application/xml"
Content-Transfer-Encoding: binary
--unique_boundary_123
Content-Type: application/xml; charset="UTF-8"
Content-Transfer-Encoding: binary
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Memo SYSTEM "Memo.dtd">
<Memo>
  <To>All Employees</To>
  <Subject>Map and Directions</Subject>
  <Body>Maps to company headquarters are attached.</Body>
  <ListOfAttachments>
    <Attachment>
      <URI>cid:001110.102203@oracle.com</URI>
      <Filename>largemap.jpg</Name>
    </Attachment>
    <Attachment>
      <URI>cid:001110.102211@oracle.com</URI>
      <Filename>detailmap.jpeg</Filename>
    </Attachment>
  </ListOfAttachment>
</Memo>
--unique_boundary_123
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <001110.102203@oracle.com>
[... Raw JPEG Image ...]
--unique_boundary_123
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <001110.102211@oracle.com>
[... Raw JPEG Image ...]
--unique_boundary_123-
```

XML Property Set Functions

Siebel EAI represents XML documents using the property set format. While Siebel EAI does not always require using the property set format, this representation is used by EAI Business Services such as the EAI XML Converter. The functions described in this section provide a simple interface for manipulating XML documents using the property set format.

High-Level Property Set Functions

These functions are used to manipulate the high-level property set passed to the Map function.

XPSGetRootElement()

This function returns the property set representing the root element of the XML document. If the root element is not present, the system raises an error. The following information presents the parameter for this function.

Syntax

XPSGetRootElement(xmlPropSetIn)

Parameter	Description
xmlPropSetIn	The name of the property set representing the root element of the XML document.

Returns

Property set

Usage

Use this function to return the root element of an XML document.

XPSCreateRootElement()

This function creates the root element in an output XML document and returns the property set representing it. The element tag in the XML document is set to the value of the tagName argument. The following information presents the parameters for this function.

Syntax

XPSCreateRootElement(xmlPropSetOut, tagName)

Parameter	Description
xmlPropSetOut	The output property set.
tagName	The name you want to supply as the root element name in the XML document.

Returns

Property set

Usage

Use this function to create the root element of an XML document that represents a property set. Because the root element does not directly map to a component in the property set, you can give it any representative name.

As an example of how the *XPSGetRootElement()* and *XPSCreateRootElement()* functions work, consider the following XML document:

```
<?xml version="1.0"?>
<!DOCTYPE LETTER SYSTEM "letter.dtd">
<letter>
  <from>Mary Smith</from>
  <to>Paul Jones</to>
  <text>Hello!</text>
</letter>
```

The root element is <letter>. The property set for the **<letter>** element can be retrieved from the input property set using *EAIXPS_GetRootElement*, or it can be created in the output property set using *EAIXPS_CreateRootElement*.

A map function that converts a letter to a memo might start with the following code:

```
function ConvertLetterToMemo (xmlPropSetIn, xmlPropSetOut)
{
  var xmlLetter = XPSGetRootElement (xmlPropSetIn);
  var xmlMemo = XPSCreateRootElement (xmlPropSetOut, "memo");

  ... Code to fill in the 'memo' from the 'letter' ...
}
```

XML Element Accessors

These functions provide access to elements represented by property sets. The following information presents the parameter for this function.

XPSGetTagName()

Retrieves the tag name of an XML element.

Syntax

XPSGetTagName (xmlPropSet)

Parameter	Description
xmlPropSet	The output property set.

Returns

String. If xmlPropSet is null, XPSGetTagName returns `null`.

XPSSetTagName()

This function sets the tag name of an XML element. The following table presents the parameters for this function.

Syntax

XPSSetTagName (xmlPropSet, tagName)

Parameter	Description
xmlPropSet	The property set.
tagName	The name you want to supply as the current element name in the XML document.

Returns

String

XPSGetTextValue()

This function returns the text value of an XML element as a string. The following table presents the parameters for this function.

Syntax

XPSGetTextValue (xmlPropSet [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
xmlPropSet	The output property set.
defaultIfNull	Specify a value to override the null default value that results if <i>xmlPropSet</i> is null.
defaultIfEmpty	Specify a value to override an empty string ("") contained in <i>xmlPropSet</i> .

Returns

String or `null`

Usage

If *xmlPropSet* is null then null is returned. You can use the optional *defaultIfNull* and *defaultIfEmpty* arguments to override null and empty string ("") return values. An element's text value is the text between an XML element's start and end tags, excluding child elements.

XPSSetTextValue()

This function sets the text value of an XML element. The following table presents the parameters for this function.

Syntax

XPSSetTextValue (xmlPropSet, text)

Parameter	Description
xmlPropSet	The property set.
text	A string you want inserted between start and end tags of an XML element.

Returns

Not applicable

Usage

The text argument should be a string. An element's text value is the text between the element's start and end tags, excluding child elements.

XPSGetAttribute()

This function retrieves an element's attribute of the given name and returns it as a string. The following table presents the parameters for this function.

Syntax

XPSGetAttribute (xmlPropSet, name [, defaultIfNull [, defaultIfEmpty]])

Parameter	Description
xmlPropSet	The output property set.
name	The name you want to supply as the root element name in the XML document.
defaultIfNull	Specify a value to override the null default value that results if xmlPropSet is null.
defaultIfEmpty	Specify a value to override an empty string ("") contained in xmlPropSet.

Returns

String

Usage

A null value is returned if xmlPropSet is null or the element does not have the named attribute. The optional defaultIfNull and defaultIfEmpty arguments can be used to override null and empty string ("") return values.

XPSSetAttribute()

This function sets an element attribute value. The following table presents the parameters for this function.

Syntax

XPSSetAttribute (xmlPropSet, name, value)

Parameter	Description
xmlPropSet	The output property set.
name	Attribute name.
value	String value you want to supply as the attribute value.

Returns

String

Usage

No action is taken if any of the arguments are null.

XPSGetChildCount()

This function returns the number of children of an element. The following table presents the parameter for this function.

Syntax

XPSGetChildCount(xmlPropSet)

Parameter	Description
xmlPropSet	The property set.

Returns

Number

Usage

All children of an element are also elements.

XPSGetChild()

This function returns the *n*th child element as specified by the index. The following table presents the parameters for this function.

Syntax

XPSGetChild(xmlPropSet, index)

Parameter	Description
xmlPropSet	The property set.
index	Number, starting at zero, of child elements of another element in an XML document.

Returns

Property set

Usage

Child elements are specified using a zero-based index. A value of `null` is returned if the index is invalid.

XPSFindChild()

This function returns the first child element with the `tagName`. The following table presents the parameters for this function.

Syntax

XPSFindChild (xmlPropSet, tagName)

Parameter	Description
xmlPropSet	The property set.
tagName	An XML element tag that signifies the first child element of another XML element.

Returns

Property set.

Usage

A value of `null` is returned if there is no child with the specified tag name.

XPSAddChild()

This function creates a new child element with the `tagName` and appends it to the list of `xmlPropSet`'s children. The following table presents the parameters for this function.

Syntax

XPSAddChild (xmlPropSet, tagName [, textValue])

Parameter	Description
xmlPropSet	The property set.
tagName	The name you want to give to the new child element.
textValue	Optional. Sets the text value of the new element.

Returns

Property set

Examples

The following example converts a <letter> to a <memo>.

Note: The input letter in this example is slightly different from the previous example.

The input XML document is:

```
<letter
  from="Mary Smith"
  to="Paul Jones">
  <text>Hello!</text>
</letter>
```

The conversion function converts this to a memo format, as follows:

```
<memo>
  <type>Interoffice Memo</type>
  <header>
    <from>Mary Smith</from>
    <to>Paul Jones</to>
  </header>
  <body>Hello!</body>
</memo>
```

The map function that performs this conversion is:

```
function ConvertLetterToMemo (xmlPropSetIn, xmlPropSetOut)
{
  var letter = XPSGetRootElement (xmlPropSetIn);
  var memo = XPSCreateRootElement (xmlPropSetOut, "memo");
  XPSAddChild (memo, "type", "Interoffice Memo");
  var header = XPSAddChild (memo, "header");
  XPSAddChild (header, "from", XPSGetAttribute (letter, "from"));
  XPSAddChild (header, "to", XPSGetAttribute (letter, "to"));
  XPSAddChild (memo, "body", XPSGetTextValue (XPSFindChild (letter, "text")));
}
```

EAI Value Maps

EAI Value Maps correlate Siebel data values with external data values.

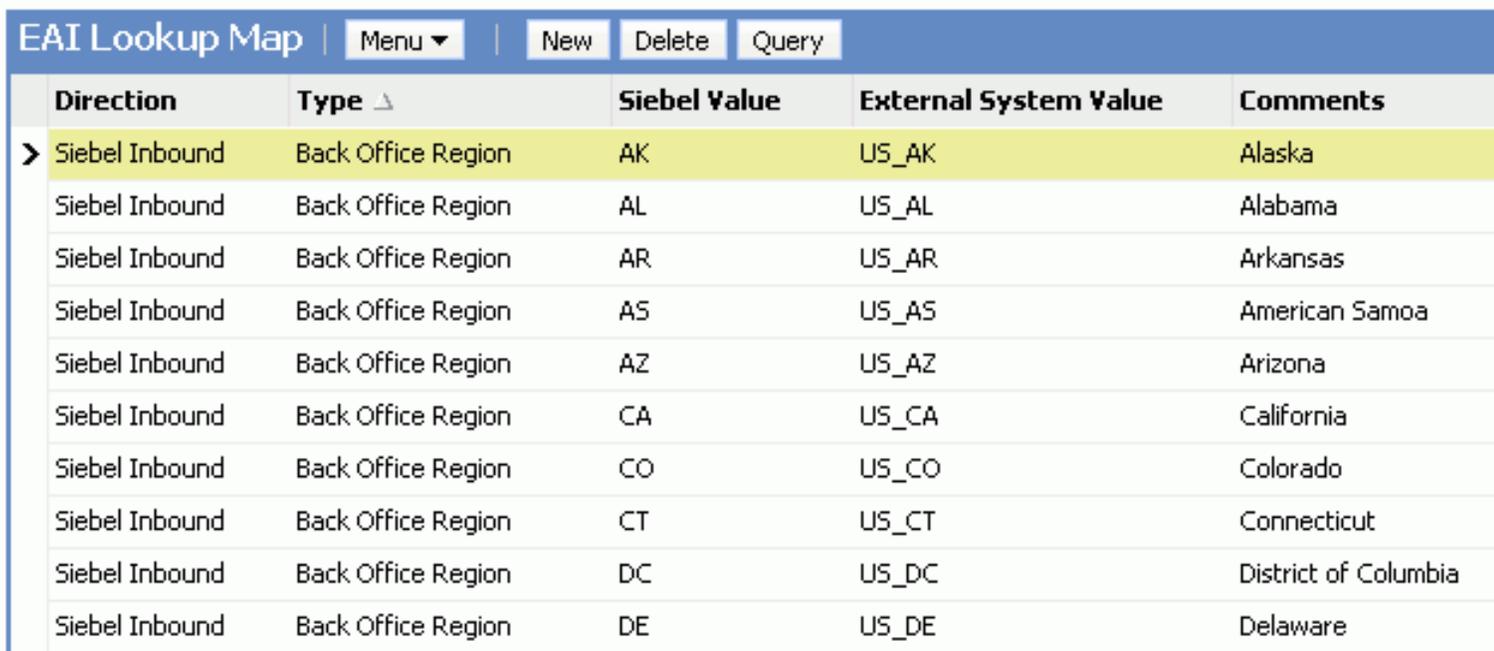
If you are:

- Sending and receiving data, you can create inbound and outbound maps for the same data
- Receiving data only, you need only to define an inbound map
- Sending data only, you need only to define an outbound map

Consider an example of how EAI Value Maps provide correlations between Siebel applications and the SAP R/3 system. SAP country codes, which are represented as two-character codes, are different from Siebel country codes, represented by the country name spelled out. An EAI Value Map provides a lookup table that lists these two sets of data side by side.

The EAI Value Map entries are stored in the EAI Value Map table. You can view and administer this table from the EAI Value Maps view in the Administration-Integration screens in the Siebel client. The Siebel client groups the entries logically based on the Type and Direction columns.

The following image shows the entries from two logical groupings, with entries for the Siebel inbound and Siebel outbound entries.



The screenshot shows the 'EAI Lookup Map' interface with a table of mappings. The table has five columns: Direction, Type, Siebel Value, External System Value, and Comments. The first row is highlighted in yellow and shows a Siebel Inbound mapping for 'Back Office Region' with Siebel Value 'AK' and External System Value 'US_AK', corresponding to 'Alaska'. The following rows show similar mappings for other states: AL (Alabama), AR (Arkansas), AS (American Samoa), AZ (Arizona), CA (California), CO (Colorado), CT (Connecticut), DC (District of Columbia), and DE (Delaware).

Direction	Type	Siebel Value	External System Value	Comments
Siebel Inbound	Back Office Region	AK	US_AK	Alaska
Siebel Inbound	Back Office Region	AL	US_AL	Alabama
Siebel Inbound	Back Office Region	AR	US_AR	Arkansas
Siebel Inbound	Back Office Region	AS	US_AS	American Samoa
Siebel Inbound	Back Office Region	AZ	US_AZ	Arizona
Siebel Inbound	Back Office Region	CA	US_CA	California
Siebel Inbound	Back Office Region	CO	US_CO	Colorado
Siebel Inbound	Back Office Region	CT	US_CT	Connecticut
Siebel Inbound	Back Office Region	DC	US_DC	District of Columbia
Siebel Inbound	Back Office Region	DE	US_DE	Delaware

The Direction field determines the direction of the mapping and is either Siebel Outbound or Siebel Inbound. In a Siebel Outbound mapping, the Siebel Value field is the lookup key; the External System Value is the translation. In a Siebel Inbound mapping, the External System Value field is the lookup key; the Siebel Value is the translation.

You can add, remove, or modify entries in the Type group on the EAI Lookup Map view in the Siebel client. The EAI_LOOKUP_MAP_TYPE list of values defines type values. You can modify the list from the Application Administration views in the Siebel client.

Note: You cannot change the values of the Direction field, which must be Siebel Outbound or Siebel Inbound.

The data transformation methods include an interface to EAI Value Maps for translating the codes of one database to another. You use the `EAIGetValueMap` function to obtain an interface to the mappings of specific Type-Direction pairs. You use the interface object's `Translate` method to find specific keys in the Type-Direction map and retrieve the translated values.

EAIGetValueMap Function

You use the following statement in your Siebel eScript code to return a value map:

```
EAIGetValueMap (type, direction [,unmappedKeyHandler])
```

This object returns a value map for translating lookup keys using the Type-Direction combination.

- The type argument is a string found in the Type field of the EAI Value Map table.
- The direction argument must be either Siebel Inbound or Siebel Outbound string values.

A call to this function returns a `CSSEAIValueMap` object.

You can use the optional `unmappedKeyHandler` argument to control the behavior of the `Translate` method when it gets keys that do not have mappings in the table. The `unmappedKeyHandler` argument can be either a literal value or a function. If you pass a literal value, it is used as the default value. Otherwise, if you pass a function, the method calls that function, then uses the value returned by the function.

The `unmappedKeyHandler` defaults to an empty string (`""`).

EAILookupSiebel Search Function

This function returns an EAI Value Map, with inbound direction that has the external value matching the value in the []. The general format for this function is as follows:

```
EAILookupSiebel ("EAI Value Type",[Source field that lookup will be based on]).
```

EAILookupExternal Search Function

This function returns an EAI Value Map, with outbound direction that has the Siebel value matching the value in the []. The general format for this function is as follows:

```
EAILookupExternal ("EAI Value Type",[Source field that lookup will be based on]).
```

CSSEAIValueMap Translate Method

The CSSEAIValueMap object has one method: Translate. The Translate method takes one argument, as follows:

Translate (key)

The Translate method looks up the key value in the EAI Value Map and returns the translated value. The EAIGetValueMap call establishes the set of mappings for the translation using the type and direction arguments. The call looks for the key in either the Siebel Value column or in the External System Value column, depending on the value of the type argument.

- If the type is Siebel Outbound, the method returns the key found in the Siebel Value column. The translated value is in the External System Value column.
- If the type is Siebel Inbound, the method returns the key found in the External System Value column. The translated value is in the Siebel Value column.
- If key is null then the return value is null.
- If key is an empty string, the lookup is performed.

If there is no mapping, an empty string is returned.

If a nonempty string does not have a mapping, the unmappedKeyHandler value specified in the call to the EAIGetValueMap function is used to determine the translation.

EAIGetValueMap unmappedKeyHandler Argument

The unmappedKeyHandler provides a flexible mechanism for handling cases where keys are not found in the EAI Value Map. In most situations, you can use literal values for defaults or you can use one of several predefined handler functions. However, you can also provide your own handler function.

The technique you use for handling unmapped values depends on the data being mapped.

Typical strategies include:

- Use the empty string as the translation.
This is the default strategy. It clears the field if the data is being imported into your Siebel application. To follow this approach, omit the unmappedKeyHandler argument or pass it as an empty string. For example:

```
var langMap = EAIGetValueMap("SAP Language","Siebel Inbound","");
```


This example looks up a nonexistent language code and returns an empty string. For example:

```
var translatedValue= langMap.Translate ("ABC"); // returns an empty string
```
- Use `null` as the translation.
This technique makes the result unspecified rather than empty. For data imported to Siebel applications, this keeps the existing value from being overridden when performing updates. Use `null` as the unmappedKeyHandler — for example:

```
var langMap = EAIGetValueMap("SAP Language","Siebel Inbound", null);
```

- Use a literal string as the translation.

Specify the string as the unmappedKeyHandler. For example:

```
var langMap = EAIGetValueMap("SAP Language","Siebel Inbound", "Unknown  
Language");
```

- Raise an error.

This may be the best strategy if the Value Map should contain mappings for every key. You can use the `EAIValueMap_NoEntry_RaiseError` function. For example:

```
var langMap = EAIGetValueMap ("SAP Language", "Siebel Inbound",  
EAIValueMap_NoEntry_RaiseError);
```

- Use the untranslated value.

The predefined function `EAIValueMap_NoEntry_ReturnLookupKey` implements this strategy. For example:

```
var langMap = EAIGetValueMap ("SAP Language", "Siebel Inbound",  
EAIValueMap_NoEntry_ReturnLookupKey);
```

Trying to look up a nonexistent language code (for example, ABC) will return the original key. For example:

```
var translatedValue = langMap.Translate ("ABC"); // returns "ABC"
```

You can also write a custom handler function. You need to write a function taking three arguments: key, type, and direction. The value your function returns is used as the translation. For example:

```
function MyUnmappedLangHandler (key, type, direction)  
{  
  return ("Unknown Language: " + key);  
}  
  
var langMap = EAIGetValueMap ("SAP Language", "Siebel Inbound",  
MyUnmappedLangHandler);  
  
// Lookup a nonexistent language code.  
  
var translatedValue = langMap.Translate ("ABC"); // returns "Unknown Language: ABC"
```

EAIGetValueMap() Method

This method retrieves objects for the required Type-Direction mapping. The following information presents the parameters for this method.

Syntax

```
EAIGetValueMap(type, direction [, unmappedKeyHandler])
```

Parameter	Description
type	Specifies the type of transformation map.
direction	A string specifying the direction of the message. The possible values are: "Siebel Inbound" "Siebel Outbound"
unmappedKeyHandler	Specifies the value to pass to the map for an unmapped key. Can be an empty string, null, a literal, or the name of a predefined function.

Returns

An object you can use to access the EAI Value Maps.

Usage

Use this method at the beginning of a script function to retrieve objects for the required Type-Direction mapping. Then call the object's Translate method to get the translation of a code from the map table as needed within the function.

Note: Providing a Type-Direction pair that does not have an entry in the EAI Value Map raises an error at the first call to the Translate method.

Exception Handling Considerations

There are three categories of errors you might encounter in the data transformation area of your integration. These categories are:

- **Siebel errors.** Errors signaled by the built-in facilities that execute a map; for example, run-time Siebel eScript errors, business service invocation errors, BusComp errors, and errors in the data transformation functions.
 - Siebel errors are fatal, terminating execution of the map immediately.
 - The business service returns an error code other than OK. No specific error code is guaranteed, and they are not intended for workflow branching. Workflow processes can branch on the indication of an error occurrence, but not on a specific code.
 - The CSSService error stack will contain useful error information. In particular, data transformation function errors will generate error stacks describing the particular error.
- **User errors.** Errors signaled in custom maps using the EAIRaiseErrorCode call. These are similar to Siebel Framework errors, except that the map developer selects the error code and uses them for workflow branching.
 - User errors are fatal, terminating execution of the map immediately.
 - The service returns the error code specified in the call to EAIRaiseErrorCode. Your workflow can branch on this code.
 - Available error codes are those in the Workflow generic error set.

- You specify the entire error text for these generic errors in the call to `EAIRaiseErrorCode`.
- You can use the function `EAIRaiseError` to raise an error without specifying a particular error code.
- **Map status flags.** The map developer can use the `SetArgument` method to set custom status information in the output property set. For example, you can use the `SetArgument` method to indicate that a required field is missing. This can be used for workflow branching, if desired. This mechanism is independent of calls made to `EAIRaiseError`.

Error Codes and Error Symbols

All errors each have an error code, which is a unique integer. A subset of errors also each have an error symbol. An error symbol is a text string that allows you to reference specific error codes in Siebel Workflow and in Siebel eScript. Errors that do not have an error symbol cannot be used for branch decisions and cannot be raised as user errors.

Error codes returned by a data transformation service may or may not have an associated error symbol. User errors will have error symbols. Currently, errors generated by Data Transformation Functions have error symbols. Errors occurring outside the data transformation framework often will not have error symbols.

Data Transformation Error Processing

This section describes how the Data Transformation Functions handle errors, and how the high-level error code returned by the data transformation business service invocation is determined.

- **Framework errors occurring outside the Data Transformation Context.** These errors are passed through without change to the `CSSService` script invocation mechanism. That mechanism takes control and returns an error of its choice. For example, if your map invokes a `BusComp` and the `BusComp` signals an error, an exception is thrown that will be ignored by the Data Mapping Engine but passed to the `CSSService` script invocation mechanism, which sets up the error state and returns an error from the business service invocation.
- **Framework errors generated by Data Transformation Functions.** These are caught by an exception handler that sets up the state in the output `PropertySet` and passes control to the `CSSEAITEScriptService` class. `CSSEAITEScriptService` sets the error code on the business service as in the state, transforming error symbols to error codes in the process. Error symbols are specific to the failure.
- **User errors.** These are processed the same way as errors generated by the Data Transformation Functions, except that you specify the error symbols and error text in your maps.

Exception Handling Functions

When writing your data transformation scripts, you can use the following functions to handle error conditions:

- `EAIRaiseError`
- `EAIRaiseErrorCode`
- `EAIFormatMessage`

Note: Before proceeding, read *Exception Handling Considerations*.

EAIRaiseError() Method

This method raises a fatal error and terminates the script. The following table presents the parameters for this method.

Syntax

EAIRaiseError(msg [, formatParameters])

Parameter	Description
msg	Error message text from the Data Mapping Engine.
formatParameters	Optional string arguments inserted in the return value in the positions specified by the positional arguments in the msg parameter. A maximum of nine format parameters are allowed.

Usage

You can provide format parameters to format the message text. For details, see [EAIFormatMessage\(\) Method](#).

EAIRaiseErrorCode() Method

This method raises a fatal error, terminates the script, and returns an error symbol that it receives from the business service.

Syntax

EAIRaiseErrorCode(errorSymbol, msg)

Usage

You can use this function when you want to pass an error symbol to a workflow as an indication to branch on an exception. If you are not branching on the specific error code in your workflow, use EAIRaiseError instead.

EAIFormatMessage() Method

This method formats strings that have position-independent arguments. The following table presents the parameters for this method.

Syntax

EAIFormatMessage(msg [, formatParameters])

Parameter	Description
msg	A string that contains positional arguments. The substitution operation replaces the percent sign followed by a digit with the corresponding format parameter.
formatParameters	Optional string arguments inserted in the return value in the positions specified by the positional arguments in the msg parameter. A maximum of nine format parameters are allowed.

Returns

A string of the formatParameters argument values in the positions specified by the positional arguments included in the msg parameter.

Usage

You can use this function to generate messages from strings that are translated and whose positions have changed as a result of the translation.

Example

```
EAIFormatMessage("Data: '%2', '%3', '%1'", "A", "B", "C")
```

returns the string:

```
"Data: 'B', 'C', 'A'"
```

Sample Siebel eScript

This section provides a sample Siebel eScript map for transforming data from a Siebel Account to SAP to retrieve an order list. The map is used to convert between Oracle's Siebel Account object and the equivalent SAP R/3 objects.

```
function GetSAPOrderStatus_SiebelToBAPI (inputMsg, outputMsg)
{
  /* Input Objects' Integration Components:
  * Order Object (Order - Get SAP Order Status (Siebel))
  * Order
  *
  * Output Object's Integration Components:
  * BAPI Import (Order - Get SAP Order Status (BAPI Input))
  * Import Parameters
  */

  /*
  * Set up EAI Lookup objects
  */

  /*
  * Set up EAI Input Message objects
  */

  var iOrderObj; // Siebel Order instance
  var iOrderComp; // Order
  /*
  * Set up EAI Output Message objects
  */

  var oGSObj; // BAPI instance
  var oGSImportComp; // Import Parameters

  /*
  * Find and create high-level integration object
  */

  iOrderObj = inputMsg.GetIntObj ("Order - Get SAP Order Status
  (Siebel)");

  oGSObj = outputMsg.CreateIntObj ("Order - Get SAP Order
```

```
Status
(BAPI Input) );
/*
 * Read int object instances from EAI message
 */

while (iOrderObj.NextInstance ())
{

/*
 * Create "Get Status" object
 */

oGSObj.NewInstance ();

/*
 * Read "Order" component
 */

iOrderComp = iOrderObj.GetPrimaryIntComp ("Order");
oGSImportComp = oGSObj.CreatePrimaryIntComp
("Import Parameters");

if (iOrderComp.NextRecord ())
{
/*
 * Write "Import Parameters" component
 */

oGSImportComp.NewRecord ();
oGSImportComp.SetCopySource (iOrderComp);
oGSImportComp.CopyFieldValue ("SALESDOCUMENT",
"Integration Id");
}
}
}
```

Conversion Between JSON and Siebel Property Sets

This topic helps you understand that the Siebel XSL to XML converter business service has two methods for converting Siebel property sets to JSON and back.

You can convert Siebel Property Sets to JSON and JSON to Siebel Property Sets using the *Siebel XSL To XML Converter* Business Service. The Business Service has two methods for this purpose.

- **pstojson** – Converts a Siebel Property Set to JSON.
- **jsontops** – Converts JSON to a Siebel Property Set.

To convert a Siebel Property Set to a JSON string there are a few rules.

Array of objects

If a set of properties in a Siebel Property Set should become a JSON array, first create a parent Property Set. The parent will contain all the child Property Sets and is the array's container. The parent Property Set has a Type with the "ListOf" prefix. After *ListOf* you specify the type of objects that will exist in the array. For instance, if you're going to have a list of *Truck* objects, the parent Property Set will have a Type of "ListOf*Truck*". Each child Property Set that will be added to the parent Property Set will have a Type of "Truck". The Business Service looks for the Property Sets with the Type that matches the ListOf<type> in its parent. In the below example where we want an array of Cars, the Type of the Property

Set (Parent) containing the instances of **Cars** has the Type: *ListOfCars*, its child Property Sets will have the type "**Cars**". Similarly, the array of **Trucks** has the Type *ListOfTrucks* its child Property Sets will have the type "**Trucks**". Each instance of a car or truck will be its own Property Set with the Type of *Cars* or *Trucks*.

Arrays of primitives

The Type of a Property Set that's meant to hold all the values of a primitive array has this format: "Listof<your type>__Primitive". There are two consecutive underscores before the word Primitive. The child Property Sets would then have a type matching <your type>. For example, if I've a Property Set with the Type *ListOfTyres__Primitive*, its child Property Sets will have the type "**Tyres**".

Each child Property Set will contain one and only one property which is the primitive value. To add another primitive value as a property, create one more child Property Set with type <your type> (*Tires*, for example) and add a property within it with its primitive value. Their property names are in the format below. (The default type is string.)

Note: Within an array, the property should be of same data type and can't have mixed types. For example, the array can be of only integers, but can't be of strings and integers.

- **string_elem__value** (for strings) There are two consecutive underscores in the string before the word "value".
- **integer_elem__value** (for integers) There are two consecutive underscores in the string before the word "value".
- **boolean_elem__value** (for boolean) There are two consecutive underscores in the string before the word "value".
- **long_elem__value** (for long) There are two consecutive underscores in the string before the word "value".
- **double_elem__value** (for double) There are two consecutive underscores in the string before the word "value".
- **float_elem__value** (for float) There are two consecutive underscores in the string before the word "value".
- **null_elem__value** (for setting the value to null in JSON) There are two consecutive underscores in the string before the word "value".

The following example represents a primitive array of type **integer**. This is converted to JSON as

```
{Tires : [4,5,6] }
```

Note the prefix "*ListOf*" and the suffix "*__Primitive*" of the parent Property Set. Also note that each child Property Set has type same as parent but without the prefix and suffix. Each has only one property named as *elem_und_undvalue* which is prefixed with the data type (**integer**) that this array should contain. Here *_und* means "underscore". There are two consecutive underscores before the string "value" in each element.

```
<ListOfTires__Primitive >  
<Tires integer_elem_und_undvalue="4"/>  
<Tires integer_elem_und_undvalue="5"/>
```

```
<Tires integer_elem_und_undvalue="6"/>
</ListOfTires>
```

Skipping a particular Property Set

If a Property Set in the hierarchy isn't valid, you can explicitly skip it by using the **__SKIP_WRAPPER** suffix. Note there are two consecutive underscores before the word SKIP and before the word WRAPPER. For example, if you've got a Property Set of Type Cars and want to skip that Property Set you'd set the Type to *Cars__SKIP_WRAPPER*.

These Property Sets are added to their parent and passed to the *Siebel XSL To XML Convertor* Business Service to get the result.

Note: The conversion doesn't maintain the order from Property Set to JSON nor from JSON to Property Set.

Example - to convert a Siebel property set to JSON

```
function convertPSToJSON(outputs:PropertySet)
{
    var LPS_convertMeToJSON:PropertySet = TheApplication().NewPropertySet();
    var LPS_Automobiles:PropertySet = TheApplication().NewPropertySet();
    var LPS_listOfTrucks:PropertySet = TheApplication().NewPropertySet();
    var LPS_truckOne:PropertySet = TheApplication().NewPropertySet();
    var LPS_truckTwo:PropertySet = TheApplication().NewPropertySet();
    var LPS_listOfCars:PropertySet = TheApplication().NewPropertySet();
    var LPS_carOne:PropertySet = TheApplication().NewPropertySet();
    var LPS_carTwo:PropertySet = TheApplication().NewPropertySet();

    var LPS_output:PropertySet = TheApplication().NewPropertySet();
    var LBS_converter:Service = TheApplication().GetService("Siebel XSL To XML Convertor");

    var LPS_ListOfTyresIntPrimitive = TheApplication().NewPropertySet();
    var LPS_TyresInt = TheApplication().NewPropertySet();
    var LPS_ListOfTyresStr1Primitive = TheApplication().NewPropertySet();
    var LPS_TyresStr1 = TheApplication().NewPropertySet();
    var LPS_ListOfTyresStr2Primitive = TheApplication().NewPropertySet();
    var LPS_TyresStr2 = TheApplication().NewPropertySet();
    var LPS_ListOfTyresStr3Primitive = TheApplication().NewPropertySet();
    var LPS_TyresStr3 = TheApplication().NewPropertySet();

    //set up the hierarchy.
    LPS_Automobiles.SetType("Automobiles");

    //ListOf is the string that tells the parser this will be an array of objects with Type = Trucks (or Cars)
    LPS_listOfTrucks.SetType("ListOfTrucks");
    LPS_listOfCars.SetType("ListOfCars");

    //These Property Sets with these specific types will be the members of the array of objects. They must have
    the same Type as their parent
    //Property Set with the "ListOf" prefix. The ListOfCars Property Set will look for a child Property Set
    with the Type "Cars".
    LPS_truckOne.SetType("Trucks");
    LPS_truckTwo.SetType("Trucks");
    LPS_carOne.SetType("Cars");
    LPS_carTwo.SetType("Cars");

    //This sets up primitive arrays. The ListOf<your type>__Primitive (note there are two underscores there) is
    the way to set the Type of these arrays.
    //Whatever you specify as <your type> is what you will use as the Type for child Property Sets.
    LPS_ListOfTyresIntPrimitive.SetType("ListOfTyres__Primitive");
    LPS_ListOfTyresStr1Primitive.SetType("ListOfTyres__Primitive");
    LPS_ListOfTyresStr2Primitive.SetType("ListOfTyres__Primitive");
    LPS_ListOfTyresStr3Primitive.SetType("ListOfTyres__Primitive");
```

```
LPS_carOne.SetProperty("make","Tesla");
LPS_carOne.SetProperty("Steering Wheel", "round");
LPS_carOne.SetProperty("Seats", "five");
LPS_carOne.SetProperty("Battery", "75KW");
LPS_carOne.SetProperty("Frunk", "empty");

//This is the actual array of the primitives. In this case they are strings so the property will have the
following format.
//string_elem_value (There are two consecutive underscores in the property name.)
//The Type of this Property Set is the <your type> from the parent Property Set.
LPS_TyresStr1 = TheApplication().NewPropertySet();
LPS_TyresStr1.SetType("Tyres");
LPS_TyresStr1.SetProperty("string_elem_value", "four");

//For each one of these primitive values, add the Property Set instance containing it to its parent
Property Set.
LPS_ListOfTyresStr1Primitive.AddChild(LPS_TyresStr1);

//For each primitive value in the array, create a new Property Set instance and assign it a value. Then add
it to the parent for
//That type.
LPS_TyresStr1 = TheApplication().NewPropertySet();
LPS_TyresStr1.SetType("Tyres");
LPS_TyresStr1.SetProperty("string_elem_value", "five");
LPS_ListOfTyresStr1Primitive.AddChild(LPS_TyresStr1);

LPS_TyresStr1 = TheApplication().NewPropertySet();
LPS_TyresStr1.SetType("Tyres");
LPS_TyresStr1.SetProperty("string_elem_value", "six");
LPS_ListOfTyresStr1Primitive.AddChild(LPS_TyresStr1);

//Once we have all the values for the array, add them to the car object.
LPS_carOne.AddChild(LPS_ListOfTyresStr1Primitive);

//add this to the parent array of cars.
LPS_listOfCars.AddChild(LPS_carOne);

//Do the same with the second instance of a car object.
LPS_carTwo.SetProperty("make","Jeep");
LPS_carTwo.SetProperty("Steering Wheel", "round");
LPS_carTwo.SetProperty("Seats", "five");
LPS_carTwo.SetProperty("GasTank", "20 Gal.");
LPS_carTwo.SetProperty("Hood", "closed");

//We are adding a single string to the array of primitives.
LPS_TyresStr2 = TheApplication().NewPropertySet();
LPS_TyresStr2.SetType("Tyres");
LPS_TyresStr2.SetProperty("string_elem_value", "five");
LPS_ListOfTyresStr2Primitive.AddChild(LPS_TyresStr2);

LPS_carTwo.AddChild(LPS_ListOfTyresStr2Primitive);

LPS_listOfCars.AddChild(LPS_carTwo);

//The cars have been handled, now do the same thing with trucks.
LPS_truckOne.SetProperty("make","Ford");
LPS_truckOne.SetProperty("Steering Wheel", "round");
LPS_truckOne.SetProperty("Seats", "five");
LPS_truckOne.SetProperty("GasTank", "20 Gal.");
LPS_truckOne.SetProperty("Hood", "closed");

LPS_TyresStr3 = TheApplication().NewPropertySet();
LPS_TyresStr3.SetType("Tyres");
LPS_TyresStr3.SetProperty("string_elem_value", "six");
LPS_ListOfTyresStr3Primitive.AddChild(LPS_TyresStr3);
```

```
lPS_truckOne.AddChild(lPS_ListOfTyresStr3Primitive);

lPS_listOfTrucks.AddChild(lPS_truckOne);

lPS_truckTwo.SetProperty("make", "Rivian");
lPS_truckTwo.SetProperty("Steering Wheel", "round");
lPS_truckTwo.SetProperty("Seats", "six");
lPS_truckTwo.SetProperty("Battery", "225KW");
lPS_truckTwo.SetProperty("Hood", "closed");

//The second Truck instance will have an array of integers, not strings.
//Its property is "integer_elem_value". There are two consecutive underscores in that property name.
lPS_TyresInt = TheApplication().NewPropertySet();
lPS_TyresInt.SetType("Tyres");
lPS_TyresInt.SetProperty("integer_elem_value", "4");
lPS_ListOfTyresIntPrimitive.AddChild(lPS_TyresInt);

lPS_TyresInt = TheApplication().NewPropertySet();
lPS_TyresInt.SetType("Tyres");
lPS_TyresInt.SetProperty("integer_elem_value", "5");
lPS_ListOfTyresIntPrimitive.AddChild(lPS_TyresInt);

lPS_TyresInt = TheApplication().NewPropertySet();
lPS_TyresInt.SetType("Tyres");
lPS_TyresInt.SetProperty("integer_elem_value", "6");
lPS_ListOfTyresIntPrimitive.AddChild(lPS_TyresInt);

lPS_truckTwo.AddChild(lPS_ListOfTyresIntPrimitive);

lPS_listOfTrucks.AddChild(lPS_truckTwo);

//Now the two arrays are added to the parent automobile array. The cars and trucks are both added here.
lPS_Automobiles.AddChild(lPS_listOfCars);
lPS_Automobiles.AddChild(lPS_listOfTrucks);

//Finally, to process the entire hierarchy, add the automobiles to the top level Property Set. This is the
one we pass
//to the converter Business Service.
lPS_convertMeToJSON.AddChild(lPS_Automobiles);

//Convert this hierarchy to JSON. The result will be in the Value of the output Property Set.
lBS_converter.InvokeMethod("pstojson", lPS_convertMeToJSON, lPS_output);

//send back the output.
Outputs.AddChild(lPS_output);
```

This is the JSON produced from this eScript.

```

{
  "Cars": [
    {
      "Tyres": [
        "six",
        "five",
        "four"
      ],
      "Frunk": "empty",
      "Battery": "75KW",
      "Steering Wheel": "round",
      "make": "Tesla",
      "Seats": "five"
    },
    {
      "Tyres": [
        "five"
      ],
      "Steering Wheel": "round",
      "make": "Jeep",
      "Hood": "closed",
      "Seats": "five",
      "GasTank": "20 Gal."
    }
  ],
  "Trucks": [
    {
      "Tyres": [
        "six"
      ],
      "Steering Wheel": "round",
      "make": "Ford",
      "Hood": "closed",
      "Seats": "five",
      "GasTank": "20 Gal."
    },
    {
      "Tyres": [
        6,
        5,
        4
      ],
      "Battery": "225KW",
      "Steering Wheel": "round",
      "make": "Rivian",
      "Hood": "closed",
      "Seats": "six"
    }
  ]
}

```

Primitive
array of string

Primitive array
of integers

Example - creating property sets from JSON

To create property sets from JSON, we can use the above output as a sample. We call the *jsontops* method of the *Siebel XSL To XML Converter* business service and supply the JSON as the **value** for the input Property Set.

For example, when this json is input as the value for the input arguments in the Business Service Simulator in the Siebel client the output is shown below.

```

{"Automobiles": {"CarModels": ["Jeep", "Tesla"], "TruckModels": ["Ford", "Rivian"], "Cars": [{"make": "
Tesla", "Steering
Wheel": "round", "Tires": ["four", "five", "six"], "Seats": "seven", "Battery": "75KW", "Hood": "closed"}
, {"make": "Jeep", "Steering Wheel": "round", "Tires": ["five"], "Seats": "five", "GasTank": "20
Gal.", "Hood": "closed"}], "Trucks": [{"make": "Ford", "Steering
Wheel": "round", "Tires": ["six"], "Seats": "five", "GasTank": "20
Gal.", "Hood": "closed"}, {"make": "Rivian", "Steering
Gal.", "Hood": "closed"}, {"make": "Rivian", "Steering
Wheel": "round", "Tires": [4, 5, 6], "Seats": "six", "Battery": "225KW", "Hood": "closed"}]}}

```

This is what's returned to the caller.

```

<?xml version="1.0" encoding="UTF-8"?><?Siebel-Property-Set EscapeNames="true"?>
<RespBody>
  <Automobiles>

```

```
<ListOfTruckModels>
<TruckModels elem_und_undvalue="Ford"/>
<TruckModels elem_und_undvalue="Rivian"/>
</ListOfTruckModels>
<ListOfCarModels>
<CarModels elem_und_undvalue="Jeep"/>
<CarModels elem_und_undvalue="Tesla"/>
</ListOfCarModels>
<ListOfTrucks>
<Trucks make="Rivian" Steering_spcWheel="round" Battery="225KW" Seats="six" Hood="closed">
<ListOfTires>
<Tires elem_und_undvalue="4"/>
<Tires elem_und_undvalue="5"/>
<Tires elem_und_undvalue="6"/>
</ListOfTires>
</Trucks>
<Trucks make="Ford" GasTank="20 Gal." Steering_spcWheel="round" Seats="five" Hood="closed">
<ListOfTires>
<Tires elem_und_undvalue="six"/>
</ListOfTires>
</Trucks>
</ListOfTrucks>
<ListOfCars>
<Cars make="Jeep" GasTank="20 Gal." Steering_spcWheel="round" Seats="five" Hood="closed">
<ListOfTires>
<Tires elem_und_undvalue="five"/>
</ListOfTires>
</Cars>
<Cars make="Tesla" Steering_spcWheel="round" Battery="75KW" Seats="seven" Hood="closed">
<ListOfTires>
<Tires elem_und_undvalue="four"/>
<Tires elem_und_undvalue="five"/>
<Tires elem_und_undvalue="six"/>
</ListOfTires>
</Cars>
</ListOfCars>
</Automobiles>
</RespBody>
```